VŠB – Technická univerzita Ostrava
Fakulta elektrotechniky a informatiky
Katedra informatiky

Simulátor vestavného procesně funkcionálního jazyka

Simulator of Embedded Process Functional Language

2011 Bc. Jan Šlahař

Prohlašuji, že jsem tuto diplomovou práci vypracoval samostatně. Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

V Ostravě 5. května 2011 .....

Rád bych na tomto místě poděkoval svému školiteli Ing. Markovi Běhálkovi, Ph.D. za možnost zúčastnit se na tomto projektu a poznání technologie WPF. A dále bych chtěl poděkovat všem, kteří mi s prací pomohli, protože bez nich by tato práce nevznikla.

Abstrakt

Požadavky na vývojový proces vestavných systému se neustále zvyšují. Agilní metodologie se obecně snaží u vývojového cyklu eliminovat různá vývojová rizika tak rychle, jak jen to jde. Řešením může být funkční model či prototyp alespoň kritických částí systému. Funkcionální jazyky jsou z tohoto pohledu velice atraktivní. Mají několik zajímavých vlastností jako: výborný mechanismus abstrakce, vytvořený kód je jednoduchý a snadno rozšířitelný. Tyto jazyky tak mohou být snadno použitelné jako nástroje produkující jakýsi spustitelný návrh. V této práci je popsána implementace simulátoru pro jazyk e-*PFL*, který obsahuje debugger pro vestavný systém. Primární určení je k vizualizaci a ladění vytvářeného systému. Lazení funkcionálních programů je obtížné a tato práce se zaměřuje na "koordinační"vrstvu.

Klíčová slova: simulator, e-PFL, WPF, kompilátor, vestavný systém

Abstract

Demands on development process of embedded systems are increasing. Agile methodologies often try to eliminate different development risks as early as possible in development cycle. Solution can be a working model or prototype of at least critical system parts. Functional languages are very attractive from this perspective. They have several interesting properties like excellent abstraction mechanism; produced code is concise and extensible. These languages can be used as a tool producing a kind of executable design. This work describes the implementation of simulator for the language e-*PFL*, which includes a debugger for embedded systems. Is primarily used for visualization and debugging designated system. Tuning of functional programs is difficult, and this work focuses on "coordination"layer.

Keywords: simulator, e-PFL, WPF, compiler, embedded system

Seznam použitých zkratek a symbolů

ePFL	–	Embedded Process Functional Language
PFL	–	Process Functional Language
PPFL	–	Parallel Process Functional Language
.NET	–	.NET Framework
HIL	–	Hardware In the Loop
MRJ	–	Motorová řídící jednotka
PC	–	Personal computer
WPF	–	Windows presentation foundation
RIA	–	Rich Internet application
BW	–	Background worker
UI	–	User interface

Obsah

1 Úvod **5**
1.1 Cíle práce . . . 5
1.2 Obsah práce . . . 5

2 Související technologie **6**
2.1 HIL . . . 6
2.2 Simulink . . . 8

3 Vestavný procesně funkcionální jazyk **10**
3.1 Koordinační vrstva . . . 10
3.2 Základní konstrukce jazyka e-*PFL* . . . 11
3.3 Praktická realizace . . . 14

4 Příklad použití e-*PFL* **16**
4.1 Popis vestavného systému pro vending machine . . . 16
4.2 UML vestavného systému . . . 16
4.3 Kompozice vestavného systému . . . 18
4.4 Výběr procesu v zařízení . . . 19
4.5 Závěr . . . 20

5 Rozšíření překladače e-*PFL* **21**
5.1 Popis překladače . . . 21
5.2 Třída Breakpoint . . . 22
5.2.1 Spuštění překladače s breakpointy . . . 22
5.2.2 Generování breakpointů z e-*PFL* . . . 22
5.3 Třída BreakpointManager . . . 23
5.4 Zdrojový funkcionální kód vestavné funkce . . . 24
5.5 Referenční funkce k vestavné funkci . . . 24
5.6 Proces generování instance simulátoru . . . 26

6 Simulátor e-*PFL* **28**
6.1 Zadání . . . 28
6.1.1 Funkční požadavky . . . 28
6.1.2 Nefunkční požadavky . . . 30
6.2 Architektura aplikace . . . 31
6.3 BackgroundWorker . . . 32

6.4 Dynamická analýza . . . 35
6.4.1 Generování vestavného systému do canvasu . . . 35
6.5 Stavová analýza . . . 36
6.5.1 Zařízení . . . 36
6.6 Statická analýza . . . 38
6.6.1 TreeNode . . . 38
6.7 Vstupní zařízení pro vestavný systém . . . 39
6.8 Návrh uživatelského rozhraní . . . 40
6.8.1 RibbonMenu . . . 41
6.8.2 Normální mód . . . 42
6.8.3 Debugovací mód . . . 44
6.8.4 Vstup pro vstupní zařízení . . . 46
6.8.5 Nastavení zařízení . . . 46
6.8.6 Nastavení breakpointů . . . 46
6.9 Technika drag-and-drop . . . 47
6.9.1 Uložení rozložení kompozice vestavného systému . . . 48
6.9.2 Načtení rozložení kompozice vestavného systému . . . 48
6.9.3 Struktura XML souboru . . . 48

7 Ukázka vygenerovaného příkladu **50**

8 Závěr **53**

9 Reference **54**

Přílohy **55**

A Zdrojové kódy **56**
A.1 e-*PFL* . . . 56
A.2 C# . . . 58
A.3 XML . . . 60

Seznam obrázků

1 HIL simulátor 7
2 Simulink simulátor 9
3 Use case diagram vending machine 16
4 Sequence diagram vending machine 17
5 Výběr procesu v zařízení 19
6 Schéma překladače e-*PFL* 21
7 Rozložení částí vestavného systému (3, 4 a 5 zařízení) 29
8 Jemné krokování zařízení 29
9 Hrubé krokování zařízení 29
10 Architektura simulátoru 32
11 Stavy zařízení 37
12 Hierarchická struktura dědění z abstraktní třídy `TreeNode` 38
13 Vstupní zařízení InCoin pro vestavný systém 40
14 RibbonMenu 42
15 Normální mód simulace 43
16 Debugovací mód simulace 45
17 Dialog pro vstupní zařízení 46
18 Dialog pro nastavení zařízení 46
19 Dialog pro nastavení breakpointů 47
20 Dialog pro výběr předmětu 50
21 Dialog pro vházení mincí 51
22 Dialog pro nastavení breakpointů 52
23 Dialog pro nastavení breakpointů 52

Seznam výpisů zdrojového kódu

1 Definice breakpoint funkce v e-*PFL* 19
2 Struktura XML konfigurace vestavného systému 23
3 Handler pro událost DoWork 33
4 Metoda StartProcess 34
5 Handler pro událost RunWorkerCompleted 35
6 Definice stromu vestavných funkcí zařízení 38
7 Struktura XML souboru rozložení kompozice vestavného systému 48
8 Definice vestavného systému vending machine 57
9 Změna vnitřního stavu zařízení 58
10 XML s rozvržením kompozice vestavného systému 60

1 Úvod

Vestavné systémy [2, 3] představují důležitou oblast informatiky. Vzhledem k různým paměťovým či výkonnostním omezením je většina těchto systémů stále realizovaná v nějakém jazyce na nižší úrovni (nejčastěji v jazyce C). Podobně jako v jiných oblastech informatiky i zde se mění požadavky na vyvíjený software. I zde se neustále zvyšují nároky například na rychlý a levnější vývoj, nebo snadnou údržbu.

1.1 Cíle práce

Před zahájením diplomové práce bylo nutné projít a pochopit syntaxi překladače e-*PFL*, seznámit se s vestavnými systémy a pochopit zdrojový kód e-*PFL*. Před samotným vývojem aplikace pro simulování bylo nutné se naučit platformu WPF 4. Zároveň s vývojem aplikace bylo nutné upravovat překladač o nové funkce vestavného systému a překladače.

Jako výsledek práce je upravený překladač, který generuje vestavný systém ze zdrojového kódu e-*PFL* do `Class library`, která se poté používá v simulátoru. Implementace simulátoru je také součástí této práce. Simulátor simuluje běh vygenerovaného vestavného systému a obsahuje debugger pro ladění koordinace a komunikace. Ladění funkcionálního kódu, který je uvnitř jednotlivých procesů se tato práce vůbec nezabývá. Simulátor se zabývá koordinací a komunikací jednotlivých jednotek a procesů, ze kterých se vestavné systému skládají.

1.2 Obsah práce

Tato diplomová práce představuje úpravu dříve existujícího překladače a realizaci simulátoru pro jazyk e-*PFL*, který je implementován jako desktopová aplikace na platformě Windows Presentation Foundation 4. Z požadavků na realizaci simulátoru bylo zapotřebí upravit a rozšířit překladač e-*PFL*.

Kapitola 2 popisuje související technologie pro simulaci vestavných systémů. Kapitola 3 popisuje vestavný procesně funkcionální jazyk, na kterém je dále založená celá tato práce. V kapitole 4 je ukázkový příklad pro simulaci vending machine (prodejního automatu), do kterého zákazník vhazuje peníze a po výběru zboží obdrží toto zboží. Kapitola 5 popisuje, které části překladače e-*PFL* bylo nutné rozšířit a které další části bylo nutné implementovat pro použití v simulátoru. Kapitola 6 popisuje realizaci simulátoru pro jazyk e-*PFL*.

2 Související technologie

Vestavné systémy se v mnoha ohledech liší od běžných aplikací [3]. Například platforma, na které je aplikace vyvíjena, je často jiná než ta, na které je výsledná aplikace provozována. Ladění může být možné pouze s využitím emulátoru. Také na vestavných systémech nebývá přítomen operační systém. Vestavné systémy jsou také vyvíjeny na různých úrovních. Mohou být vyvíjeny jako softwarové aplikace, nebo jako hardwarové komponenty, nebo také jako kombinace obou.

Na hardwarové úrovni můžeme použít technologie jako *Programmable Logic Devices* nebo VHDL (***V**ery High Speed Integrated Circuit **H**ardware **D**escription **L**anguage*). Na softwarové úrovni pak můžeme použít programovací jazyky vysoké úrovně jako je například C# a provozovat výsledný systém na platformě jako je .NET Micro Framework [2].

Testování a simulace vestavných systémů se také dá provádět na hardwarové nebo softwarové úrovni.

Na hardwarové úrovni se používá HIL simulace. Kdy se ke skutečnému systému připojí simulátor, který v reálném čase simuluje příslušné elektromechanické prostředí. Díky této metodě lze v laboratorních podmínkách otestovat chování a provozní vlastnosti systému.

Pro simulaci na softwarové úrovni se používá simulátor pro vestavné systémy Simulink, který byl primárně vyvinut k modelování vestavných systémů na hardwarové úrovni.

2.1 HIL

Metoda HIL (Hardware–In–the–Loop) se obecně používá pro simulaci elektromechanických systémů, které jsou řízeny elektronickou řídící jednotkou. Typickým příkladem takového systému v oblasti automobilů je motor, který je ovládán motorovou řídící jednotkou (MRJ). Proces, kdy se ke skutečné MRJ připojí simulátor, který nahradí veškerá elektrická čidla a akční cleny v motoru. Simulátor v reálném čase simuluje příslušné elektromechanické prostředí (motor, převodovka, podvozek, . . .). Díky této metodě lze v laboratorních podmínkách otestovat chování a provozní vlastnosti zkoušených řídících jednotek.

Celý systém se skládá z několika zařízení. Nejdůležitější z nich je vlastní simulátor, ve kterém probíhá simulace daného procesu v reálném čase. Simulátor obsahuje výkonný napájecí zdroj, regulovatelný modelem, procesorovou kartu s modulem pro komunikace s PC. Dále dle aplikace obsahuje jednu nebo více vstupně – výstupních karet, simulující elektrické signály pro řídící jednotku. Důležitá část simulátoru je blok pro umělé zátěže,

kde je možno k jednotlivým signálům od řídící jednotky připojit umělou zátěž, simulující připojené čidlo či akční člen. Součástí tohoto bloku je jednotka simulace chyb, která umožňuje zkratovat vybrané signály k napájecímu napětí, zemi nebo tyto signály rozpojit. K simulátoru, resp. jeho I/O vodičům je připojena testovaná řídící jednotka, případně další nezbytné komponenty, jako jsou palubní přístroj, škrtící klapka, atd. Tyto přídavné díly se připojují v případe, že jejich simulace je velice náročná a je jednoduší použít skutečné prvky.

Simulátor je ovládán z PC, jež je připojeno optickým kabelem. Model simulovaného procesu je vytvořen v ovládacím PC v programu MATLAB – SIMULINK. Jakmile je model funkční, je přeložen do kódu, umožňující běh aplikace v reálném čase v procesoru simulátoru. Pro tyto úkony je nutné použít toolbox Real-Time-Workshop plus příslušné překladače. Vygenerovaný kód se po optickém kabelu nahraje do simulátoru, kde již může běžet zcela autonomně bez zásahu ovládacího PC. V praxi je ovšem žádoucí, aby obsluha mohla zasahovat do procesu simulace, nebo se alespoň informovat o právě probíhajících procesech. K tomuto účelu je možné spustit na PC program Controldesk, pomocí kterého lze provádět on-line zásahy do simulace, běžící v reálném čase. Takto se prověří všechny režimy a stavy řídicích jednotek bez rizika zranění nebo zničení nákladného prototypu.

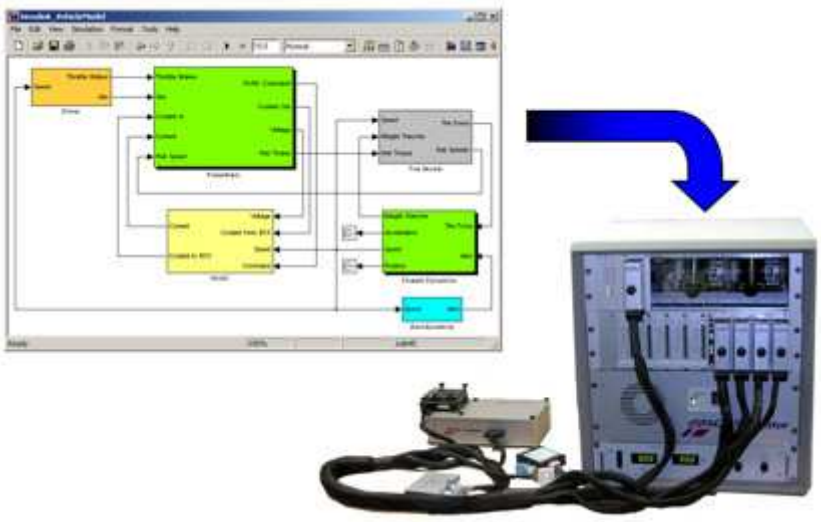

Figure 1: HIL simulátor

2.2 Simulink

Simulink je prostředí pro simulaci a multidoménový Model-Based Design pro dynamické a vestavné systémy. Poskytuje interaktivní grafické prostředí a přizpůsobitelné knihovny bloků, které umožňují navrhnout, simulovat, implementovat a testovat různé systémy, včetně komunikace, řízení, zpracování signálu, zpracování videa a zpracování obrazu. Add-on prvky rozšiřují Simulink do více domén modelování, stejně jako poskytuje nástroje pro návrh, implementaci, ověřování a validace.

Simulink je integrován s MATLABem a poskytuje okamžitý přístup k širokému spektru nástrojů, které umožňují rozvinout algoritmy, analyzovat a vizualizovat simulace, vytvářet skripty dávkového zpracování, přizpůsobovat modelovací prostředí, definovat signál, parametry a testovací data.

Klíčové vlastnosti:

- Rozsáhlé a rozšiřitelné knihovny předdefinovaných bloků.
- Grafický editor pro sestavování a řízení intuitivních blokových schémat.
- Schopnost zvládat složité vzory segmentací modelů do hierarchie komponent.
- Model Explorer k procházení, vytváření, konfigurování a hledaní všech signálů, parametrů, vlastností a vygenerovaný kód související s vytvořeným modelem.
- API, které umožní spojení s ostatními simulační programy a začlenění ručně psaného kódu.
- Režimy simulace (Normal, Accelerator a Rapid Accelerator) pro běh simulace.
- Grafický debugger a profiler pro kontrolu výsledků simulace s možností diagnostikovat výkon a neočekávané chování v návrhu.
- Plný přístup k MATLABu pro analýzu a vizualizaci výsledků, přizpůsobení modelovacího prostředí, definování signálu, parametrů a údaje ze zkoušek.
- Modelové analýzy a diagnostické nástroje pro zajištění konzistence modelu a určení chyb v návrhu.

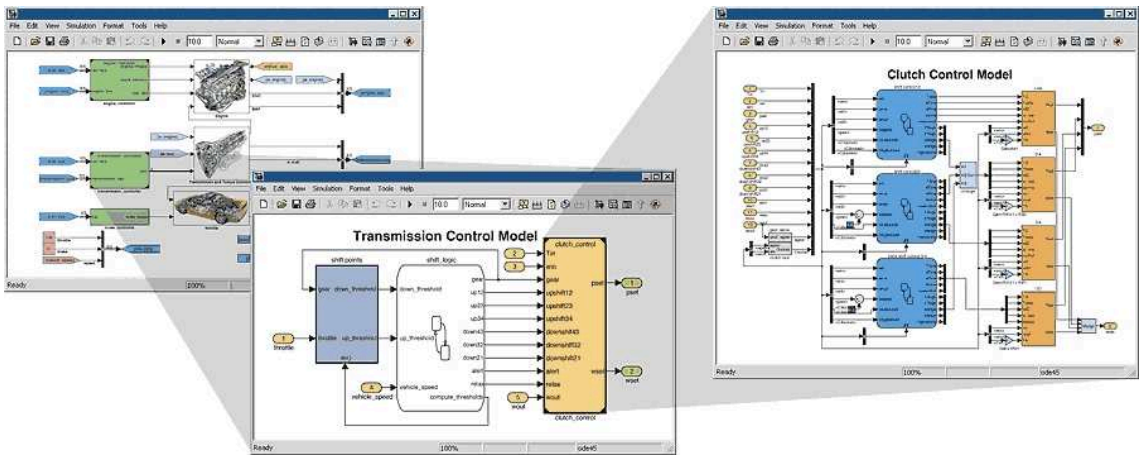

Figure 2: Simulink simulátor

Kromě standardních úloh dovoluje Simulink rychle a přesně simulovat i rozsáhlé systémy s efektivním využitím paměti počítače. Pomocí Simulinku a jeho grafického editoru lze vytvářet modely lineárních, nelineárních, v čase diskrétních nebo spojitých systémů pouhým přesouváním funkčních bloků myší. Simulink také umožňuje spouštět určité části simulačního schématu na základě výsledku logické podmínky. Tyto spouštěné a povolované subsystémy umožňují použití programu v náročných simulačních experimentech. Samozřejmostí je otevřená architektura, která dovoluje uživateli vytvářet si vlastní funkční bloky a rozšiřovat již tak bohatou knihovnu Simulinku. Hierarchická struktura modelů umožňuje koncipovat i velmi složité systémy do přehledné soustavy subsystémů prakticky bez omezení počtu bloků. Simulink, stejně jako MATLAB, dovoluje připojovat funkce napsané uživateli v jazyce C. Vynikající grafické možnosti Simulinku je možné přímo využít k tvorbě dokumentace. Mezi neocenitelné vlastnosti Simulinku patří nezávislost uživatelského rozhraní na počítačové platformě. Přenositelnost modelů a schémat mezi různými typy počítačů umožňuje vytvářet rozsáhlé modely, které vyžadují spolupráci většího kolektivu řešitelů na různých úrovních.

3 Vestavný procesně funkcionální jazyk

Jazyk lze použít pro modelování vestavných systémů na vysoké úrovni abstrakce, stále však jde o programovací jazyk, jehož výstupem je spustitelný kód.

Při vývoji jazyka byl použit jazyk *PPFL* a to zejména na implementační úrovni. Jak postupně vznikala koncepce jazyka e-*PFL*, tak se v mnoha ohledech oddělil od jazyka *PFL*. Vytvořený jazyk tak integruje například principy použité u jiných funkcionálních jazyků. Základní vlastnosti e-*PFL* budou popsány v této kapitole.

3.1 Koordinační vrstva

Většina funkcionálních jazyků uzpůsobených pro implementaci vestavných systémů používá dvouúrovňový přístup k návrhu jazyka. Na jazykové úrovni je obvykle čistě funkcionální vrstva (sloužící k popisu výkonné částí systému) obalena do koordinační vrstvy. Tato vrstva pak řeší problémy spojené s komunikací. Takto jsou funkcionální programovací jazyky často rozšířeny o jazykové konstrukce, které mají vedlejší efekty. Jazykové konstrukce řeší v principu dva problémy a to vytváření nějakých souběžných procesů a jejich komunikaci. Jako příklady mohou sloužit dva jazyky, představující určité extrémy při řešení koordinační vrstvy.

- Embedded Gofer [4] — tento jazyk umožňuje vytvářet další paralelní či souběžné procesy. K jejich popisu používá monády. Komunikace je řešena pomocí předávání zpráv. Programovací jazyk je rozšířen o základní konstrukce, které posílání zpráv umožňují. Toto řešení dává programátorovi možnost pracovat s koordinační vrstvou a řídit dynamicky topologii vytvářeného systému. Problémem může být, že řadu vlastností vytvářeného systému nejsme schopni nijak určit.

- Hume [9, 13] — prezentuje jiný přístup. Zde je struktura systému na koordinační úrovni definována staticky přímo ve zdrojovém kódu. Programovací jazyk obsahuje jakási makra, pomocí kterých můžeme tuto strukturu před vlastní kompilací měnit, ale v principu je v době kompilace struktura vytvářené aplikace daná. Na druhou stranu právě díky tomu jsme schopni v době kompilace zjistit řadu zajímavých informací o výsledném systému. Například můžeme vypočítat některá běhová omezení.

Při vytváření nového jazyka muselo být zvoleno, jak bude vypadat koordinační vrstva.

Jazyk e-*PFL* umožňuje koordinační vrstvu definovat dynamicky. Jazyk obsahuje konstrukce, které umožňují definici nových funkčních jednotek a dynamicky měnit jejich propojení. Na druhou stranu má statický modle na koordinační úrovni své výhody.

Primárním určením jazyka e-$\mathcal{PFL}$ je, že slouží jako základ pro vývoj modelovacího nástroje. Statický model koordinace (alespoň od nějakého bodu v čase) je zajímavý. Například je lehčí jeho vizualizace. Také ulehčuje využití jiných technologií na nižších úrovních. Například při použití jazyka Hume, který má statický model koordinace, potřebujeme realizovaný systém od jisté chvíle zjednodušit a vlastně definovat na statické úrovni. Statický model také lépe postihuje některé vestavné systémy, zejména ty, které jsou řešeny i na *hardwarové* úrovni.

Při tvorbě e-$\mathcal{PFL}$ tak byl zvolen jistý kompromis. Jazyk obsahuje konstrukce, které umožňují dynamicky vytvářet realizovaný systém na koordinační úrovni. Model na koordinační úrovni tak postupně vzniká jako produkt vykonávání programu. Jeho tvorba je tak jednodušší. Jakmile je však nějaká funkční jednotka spuštěna, nelze její činnost již měnit. Model, který takto vznikne, pak již zůstává statický.

Prakticky je tento přístup řešen tak, že prvním krokem při generování cílového kódu, nebo při simulaci je *konfigurační běh*. Kdy je aplikace v e-$\mathcal{PFL}$ částečně vyhodnocena [14] a jako výsledek dostaneme strukturu aplikace na koordinační vrstvě. Struktura systému je pak uložena ve formátu XML.

Vyvíjený systém je tak striktně rozdělen na části. Zdrojové kódy v e-$\mathcal{PFL}$ popisují logiku vyvíjené aplikace. S pomocí částečného vyhodnocení je pak vytvořená konkrétní architektura vyvíjeného systému. K jednomu programu v e-$\mathcal{PFL}$ je možné vygenerovat několik různých konfigurací a tak získat několik konkrétních modelů. Na základě konfigurace pak kompilátor vygeneruje cílový kód (například v jazyce Hume) a ten lze nasadit na skutečná zařízení.

3.2 Základní konstrukce jazyka e-$\mathcal{PFL}$

V této kapitole jsou stručně popsány základní jazykové konstrukce e-$\mathcal{PFL}$. Vytvořený jazyk používá vyhodnocování na základě hodnoty. Syntaxe a sémantika vychází z $\mathcal{PFL}$ (ten zase vychází z čistě funkcionální podmnožiny jazyka Haskell). Do jazyka e-$\mathcal{PFL}$ pak byly přidány konstrukce pro modelování vestavných systémů.

Prvním krokem při vývoji jazyka e-$\mathcal{PFL}$ byla definice vestavných procesů. Vestavné procesy svou koncepcí vycházejí z procesů v $\mathcal{PFL}$(zejména na jazykové úrovni). Svou koncepcí se ale liší. Vestavné procesy popisují jednu operaci s jasně definovanými vstupy a výstupy. Funkční jednotky, ze kterých je pak skládán výsledný vestavný systém, jsou složeny právě z vestavných procesů.

Jak již bylo uvedeno v předcházejících kapitolách, je v e-$\mathcal{PFL}$ vestavný systém modelován jako soustava komunikujících funkčních jednotek. V jazyce e-$\mathcal{PFL}$ je funkční

jednotka nazvaná – `Device`. Jde o běžný datový typ definovaný v základním modulu `Prelude` (tento modul obsahuje základní knihovní funkce).

```
data Device = Process EmbProcess
            | Fair    [Device]
            | Unfair  [Device]
```

Definované jednotky (`Device`) jsou striktně složeny z vestavných procesů. Vestavné procesy jsou popsány datovým typem `EmbProcess`. Tento datový typ byl použit ve předchozí definici typu `Device`. Tento typ není definován běžným způsobem a jeho podpora je vestavěna přímo do kompilátoru. Jde o jedno z mála nestandardních rozšíření na úrovni jazyka (ostatní rozšíření jsou většinou běžné datové typy). Tyto vestavné procesy pak postihují problémy spojené komunikací na koordinační vrstvě. Syntaxe vychází ze syntaxe procesů v jazyce $\mathcal{PFL}$ a tak je blízká k definici funkce. Definice vestavného procesu je rozšířená o proměnné. Tyto proměnné jsou spojeny s parametry a také s návratovou hodnotou. Pokud je návratovou hodnotou n-tice, musí být každý její element spojen s nějakou proměnnou. Definici vestavného procesu ukazuje následující příklad.

```
work :: a Int −> b Int −>(c Int, d Int)
work x y = (x, x+y)
```

Použité proměnné reprezentují komunikační kanály. Vestavný proces tak definuje konkrétní operaci s jasně definovaným vstupem (proměnné `a` a `b` v předchozím příkladě) a výstupem (proměnné `c` a `d`). Vstup a výstup definovaných funkčních jednotek tak je určen vestavnými procesy, které obsahuje. Každá proměnná pak může být použita jako vstup právě jedné jednotky a také jako výstup právě jedné jednotky.

Vestavné procesy nemohou používat původní procesy jazyka $\mathcal{PFL}$. Vestavné procesy takto postihují problémy spojené s koordinační vrstvou, a zaobalují konstrukce mající vedlejší efekty (spojené s komunikací a synchronizací).

Namodelované funkční jednotky jsou spouštěny pomocí nativní funkce `startDevice`. Tato nativní funkce je deklarována v základním modulu `Prelude`. Symbol `()` pochází z jazyka $\mathcal{PFL}$ a reprezentuje řídící hodnotu [15].

```
startDevice :: Device −> EmbSystem −> [Annotation] −> ()
```

Každá taková funkční jednotka je autonomní systém, který pracuje v principu asynchronně a nezávisle na ostatních jednotkách. Jak bylo uvedeno, každá jednotka je vlastně složená z vestavných procesů. Když je jednotka nastartována, snaží se vykonávat vestavné procesy, které obsahuje. V jedné chvíli může být vykonáván maximálně jeden proces. Vestavné procesy mezi sebou soutěží o to, který bude vykonán. Pokud aktuálně

není vykonáván žádný vestavný proces, je vybrán nový kandidát a to na základě dostupnosti vstupu a *férovosti*. Při výběru musí být nejdříve splněna podmínka, že všechny vstupní komunikační kanály vestavného procesu obsahují hodnotu. Pokud tomu tak je, je vestavný proces schopen běhu. Mezi vestavnými procesy schopnými běhu je pak následně vybrán jeden. Jeho výběr je ovlivněn datovými konstruktory `Fair` (jsou vybírány elementy na nižší úrovni tak, aby každý potomek ve stromě měl stejnou možnost být vybrán) a `Unfair` (elementy na nižší úrovni stromu jsou vybírány podle pořadí v původní definici).

Vstupní hodnoty, reprezentované komunikačními kanály ztotožněnými s parametry vestavného procesu, jsou procesem při jeho provedení *zkonzumovány*. Po provedení se vestavný proces snaží do výstupu, komunikačních kanálů spojených s návratovou hodnotou, zapsat vypočítané výsledky. Dokud se mu to nepovede, nemůže být výpočet ukončen a nemůže se začít s vykonáváním dalšího vestavného procesu. Hodnotu lze do komunikačního kanálu zapsat ve chvíli, kdy neobsahuje žádnou hodnotu.

Jednou spuštěné funkční jednotky nelze zastavit ani modifikovat. Pro úplnost je nutné dodat, že vestavné procesy nelze používat běžným způsobem, nelze je tedy aplikovat přímo na nějaké konkrétní hodnoty.

Funkční jednotky mohou být rozděleny do komponent. Komponenty jsou definovány pomocí datového typu `EmbSystem` (typ druhého parametru funkce `startDevice`). Tento datový typ je definován v modulu `Prelude` a jeho definice vypadá následovně.

```
data Mediator = Hume | MicroNET
data EmbSystem = EmbComponent [Character] Mediator | Emulator
```

Datový typ `EmbSystem` má dva datové konstruktory. Konstruktor `Emulator` postihuje komponentu, pro kterou nebude generován cílový kód, která bude použita jen při simulaci. Druhý datový konstruktor je `EmbComponent`. Tento datový konstruktor má dva parametry. První umožňuje definovat jméno komponenty (jde o řetězec), druhý pak definuje použitého prostředníka.

V našem přístupu není generován jeden cílový kód. Programátor může modelovaný systém rozdělit na části. Každá část je pak spojená s jakýmsi prostředníkem. Pro každou z těchto komponent je pak vygenerován cílový kód. Tento cílový kód je pak vytvořen s přihlédnutím k tomuto prostředníku. Může mít tak různé vlastnosti nebo různé možnosti nasazení.

Při spouštění funkčních jednotek je také možné změnit některé jejich vlastnosti. Tyto změny můžeme provést prostřednictvím jakýchsi *anotací*. Tyto anotace jsou spojeny s konfiguracemi a konfiguračním během (ten byl popsán v sekci 3.1). Tyto anotace pak ovlivňují, jaký cílový kód bude produkován. Použitím anotací jsme schopni změnit propojení

systému na koordinační vrstvě, nastavení iniciálních hodnot nebo úroveň optimalizace cílového kódu.

Anotace jsou definovány s použitím datového typu `Annotation`. Programátor nemusí používat tyto anotace přímo. Může používat například definované funkce. Příkladem může být funkce `rename`. Ta je v modulu `Prelude`. Lze ji využít k přejmenování některého ze vstupů či výstupů. Takto lze změnit propojení jednotlivých jednotek.

```
data Attribute  = CAttribute  [Character] [Character]
data Annotation = CAnnotation [Character] [Attribute]

rename::[Character]->[Character]->Annotation
rename x y = CAnnotation "rename" [(CAttribute "old" y), (CAttribute  "new" x)]
```

Důležitým aspektem, který musel být vyřešen při praktické implementaci jazyka e-$\mathcal{PFL}$ je komunikace. Komunikace je v e-$\mathcal{PFL}$ implicitní. Spuštěné funkční jednotky komunikují prostřednictvím použitých komunikačních kanálů. Každý takový komunikační kanál ve své podstatě spojuje právě dva zdroje. Jak již bylo uvedeno, konkrétní komunikační kanál může sloužit jako vstup maximálně jedné jednotce a jako výstup také maximálně jedné jednotce. Pokud není spojen s nějakou funkční jednotkou, může být použit jako výstup například do nějakého síťového proudu, nebo například na standardní výstup. Také může modelovat konkrétní vstup systému například z nějakého senzoru.

Vzhledem k tomu, že je komunikace v e-$\mathcal{PFL}$ implicitní, jsou všechny problémy spojené se synchronizací a serializací dat řešeny v kompetenci překladače respektive použitého běhového prostředí. Kompilátor je v době překladu schopen určit, jaký je typ přenášených dat. Jde o typ svázaný s komunikačním kanálem při jeho definici. Pokud jsou spojeny dva zdroje, jejich typy musí souhlasit. Díky tomu jsme schopni vyřešit jak problémy serializace, tak problémy synchronizace. Synchronizace je v principu spojena s činností funkčních jednotek. Kompilátor musí zajistit, aby nová hodnota byla do nějakého komunikačního kanálu umístěna až ve chvíli, kdy stará hodnota byla zkonzumována a povolit jen výběr hodnoty z komunikačních kanálů, které obsahují nějakou hodnotu.

Nastavení vlastností komunikace může programátor ovlivnit v rámci konfigurace systému změnami v konfiguračním souboru.

3.3 Praktická realizace

Pro podporu programování ve vytvořeném jazyce e-$\mathcal{PFL}$ jsme implementovali několik nástrojů [1]. Základním nástrojem je překladač jazyka e-$\mathcal{PFL}$. Tento překladač vychází

[1] Implementované nástroje jsou dostupné na: `http://www.cs.vsb.cz/behalek/epfl`

z překladače jazyka *PPFL*. Překladač byl v první fázi upraven tak, aby podporoval jazykové konstrukce e-*PFL*. Jde zejména o podporu vestavných procesů. Také z něj byly (jak pokračoval výzkum) postupně odstraňovány nepotřebné konstrukce z *PPFL* respektive *PFL*. Aktuálně zbyla podpora *PFL* procesů a proměnných prostředí. Ty slouží k naplnění počátečních hodnot spouštěných funkčních jednotek. V jazyce zbyla také řídící hodnota — `()`.

Implementovaný kompilátor se však liší v oblasti generování cílového kódu. V principu má dva módy. V prvním módu musí připravit konfigurační běh. Zde je použito podobného modelu, jako při kompilování jazyka *PPFL*. Vstupní program v jazyce e-*PFL* je transformován do kódu v jazyce C#. Tento kód je pak sloučen s běhovým prostředím (to také vychází z běhového prostředí pro *PPFL*). Nyní lze vytvořené kódy v jazyce C# zkompilovat a spustit. Po spuštění je proveden konfigurační běh, budovaná aplikace je částečně vyhodnocena a je vygenerována konkrétní konfigurace. Tato konfigurace je uložena do souboru ve formátu XML. Druhý mód je klasické generování cílového kódu. Zde jsou použity jak zdrojové kódy v e-*PFL*, tak připravená konfigurace. Výsledkem generování může být kód pro simulátor, nebo cílový kód pro definované komponenty.

Kompilátor je stále vyvíjen a rozšiřován. Ne všechny popsané konstrukce byly plně integrovány. Například aktuálně je podporovány pouze synchronizace základních datových typů. Byly také realizovány další podpůrné nástroje. Například nástroj umožňující práci s vygenerovanými konfiguracemi.

4 Příklad použití e-$\mathcal{PFL}$

V této kapitole je na konkrétním příkladu demonstrováno použití jazyka e-$\mathcal{PFL}$. Příklad modeluje zjednodušený vestavný systém pro vending machine (prodejního automatu). Z tohoto příkladu dále vychází simulátor, který je popsán v následující kapitole 6.

Důležité: tento příklad je pouze pro znázornění a nejedná se o reálný model vestavného systému.

4.1 Popis vestavného systému pro vending machine

Vestavný systém pro vending machine je zjednodušený kvůli přehlednosti a složitosti od zdrojového kódu e-$\mathcal{PFL}$ až po vygenerovanou kompozici v simulátoru. Zákazník může interagovat se systémem pouze tím, že si vybírá zboží a vhází mince do automatu. Funkce jako stornování a vrácení mincí není modelovaným systémem podporována. Z tohoto důvodu obsahuje vestavný systém méně funkčních jednotek, vestavných procesů a komunikačních kanálů.

4.2 UML vestavného systému

Následující Use case diagram obr. 3 a Sekvenční diagram obr. 4 modelují funkce vestavného systému.

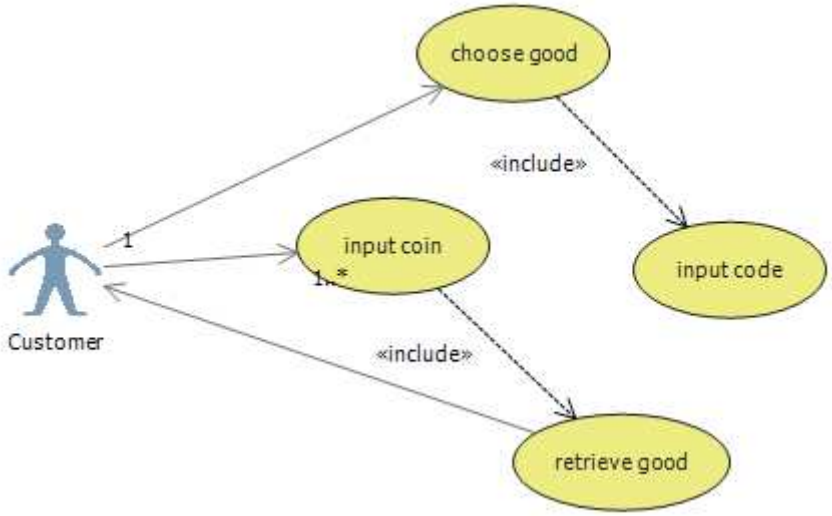

Figure 3: Use case diagram vending machine

Figure 4: Sequence diagram vending machine

4.3 Kompozice vestavného systému

Vestavný systém se skládá z jedné komponenty, která používá .NET Micro Framework. Komponenta obsahuje pět funkčních jednotek.

1. **Printer** — Reprezentuje display automatu, který informuje zákazníka. Obsahuje vestavné funkce:

   (a) ***showText1* —** zobrazuje obsah kominukačního kanálu `displayInput1`. Informuje uživatele kolik vybraný předmět stojí.

   (b) ***showText2* —** zobrazuje obsah kominukačního kanálu `displayInput2`. Informuje uživatele o stavu předmětu.

   (c) ***showText3* —** zobrazuje obsah kominukačního kanálu `displayInput3`. Informuje uživatele, jakou minci vhodil do automatu. Po vhození celé částky je platba přijata.

2. **Controller** — Reprezentuje jednotku, která se stará o vybrání produktu, který si vybral zákazník . Obsahuje vestavné funkce:

   (a) ***selection* —** Vybírá zvolený produkt zákazníkem.

   (b) ***served* —** Připravuje vybraný produkt na budoucí prodej.

   (c) ***controllerProcess* —** Pokud zvolený produkt není k dispozici, upozorní o tomto stavu zákazníka. Jinak vybírá produkt.

3. **Serving** — Reprezentuje jednotku, která se stará o vybraný produkt po jeho zaplacení. Obsahuje vestavné funkce:

   (a) ***serve* —** připraví vybraný produkt.

   (b) ***paid* —** po vhození dostatečné částky pro produkt prodá tento produkt.

4. **Counter** — Reprezentuje váhu mincí a rozpoznává nominální hodnotu mincí. Obsahuje vestavné funkce:

   (a) ***initCounter* —** rozpoznává nominální hodnotu mincí, které zákazník postupně vhazuje do automatu.

   (b) ***count* —** sčítá nominální hodnotu všech mincí, které zákazník vhodil do automatu.

5. **Memory** — Slouží jako databáze systému, kterou reprezentuje trojice `Integerů`. První člen je pořadové číslo (kód) produktu, které si volí zákazník. Druhý člen je počet disponibilních kusů v automatu. Třetí člen je nominální hodnota produktu. Obsahuje vestavné funkce:

   (a) ***getItem*** **—** získává zvolený produkt z databáze.

   (b) ***removeItem*** **—** odstraňuje zvolený produkt z databáze.

K vestavnému systému je dále navázána jedna breakpoint funkce, která při vhození mince s nominální hodnotou 50 Kč vyvolá zastavení simulace vestavného systému ve vygenerovaném simulátoru.

```
test1 :: coin Integer->Bool test1 coin = if coin == 50 then False else True
```

Výpis 1: Definice breakpoint funkce v e-*PFL*

Definice celého vestavného systému se nachází v příloze A jako výpis 8.

4.4 Výběr procesu v zařízení

Každé zařízení vestavného systému obsahují své vestavné procesy. Tyto procesy jsou uspořádány do stromové struktury. Podle příslušných vstupů (které jsou výstupy jiných zařízení) pro dané zařízení se vybírá proces, který je následně vykonáván. Po vykonání procesu je vyprodukován výstup, který je následně konzumován dalším zařízením, které tento vstup přijímá.

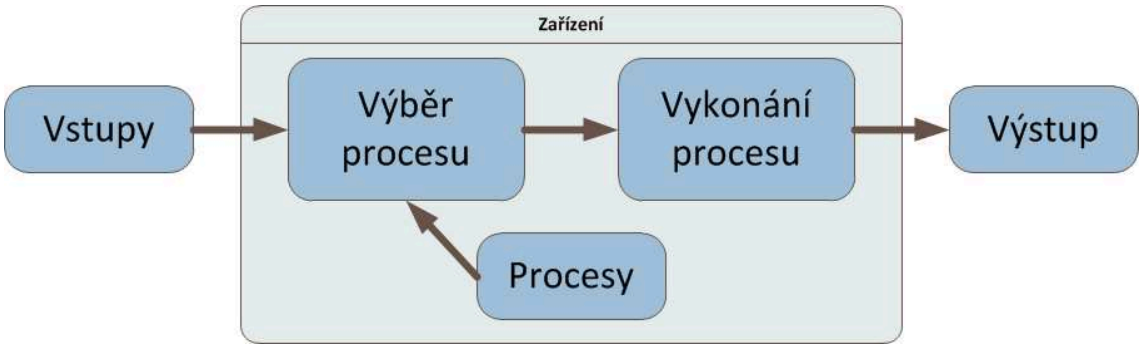

Figure 5: Výběr procesu v zařízení

4.5 Závěr

Obecně lze zdrojový kód v e-*PFL* rozdělit do dvou částí. Při definici funkčních jednotek vlastně definujeme funkcionalitu výsledného systému. Funkční jednotky takto představují základní stavební jednotky modelovaného systému. Na nižší úrovni jsou pak složeny z vestavných procesů. Zbytek programu pak definuje části modelovaného systému (spojené s generováním cílového kódu) a startovních podmínek. S využitím konfigurací systémů, které můžeme získat na základě částečného vyhodnocení ze zdrojových kódů, pak můžeme extrahovat různé statické modely vytvářeného systému na koordinační vrstvě. Díky tomu můžeme prezentovat funkcionalitu systému (například zákazníkovi), také je tento systém připraven na budoucí vizualizaci či automatické generování kódu (podobně jako nástrojů pro návrh GUI). Díky tomu, že systém je stále popsán pomocí vykonatelného programovacího jazyku, můžeme činnost modelovaného systému snadno simulovat.

Z hlediska generování cílového kódu využívá e-*PFL* na nižších úrovních jiné technologie určené pro vývoj vestavných systémů. Komponenta používá vývojový rámec .NET Micro Framework. Pro jeho podporu bylo vytvořeno jednoduché běhové prostředí. Toto běhové prostředí obsahuje základní konstrukce nutné pro běh kódu, který vygeneruje kompilátor e-*PFL*. Překladač *e-PFL* vygeneruje kód v jazyce C#. Pokud jej spojíme s vytvořeným běhovým prostředím, můžeme jej například nasadit pomocí nástrojů, které dodává společnost Microsoft. Nasazení aplikace pro vestavné systémy nemusí být triviální. Komplexní nástroje dodávané k vývojovému rámci .NET Micro Framework tak mohou být velkou pomocí.

Takto můžeme eliminovat různá technická rizika spojená s vývojem vestavného systému. Také můžeme těžit z výhod jiných nástrojů a technologií. Celý systém nebo jeho části lze později realizovat na hardwarové úrovni nebo přepsat s použitím jiného jazyka jako je C. Například proto, abychom snížili spotřebu energie, nebo urychlili běh systému. I pak může být vytvořený model užitečný. Může sloužit například jako jistý druh spustitelné dokumentace. Funkcionální jazyky jsou v mnoha ohledech sebe-popisné a nepotřebují další rozsáhlou dokumentaci.

5 Rozšíření překladače e-*PFL*

Cílem této kapitoly bylo rozšířit dosavadní implementaci překladače pro generování cílového kódu do instance simulátoru (generování celé solutiony), který je popsán v Kapitole 6. Překladač byl implementován v jazyce C# na platformě .NET Framework 3.5 a pro kompilování simulátoru, který je na platformě .NET Framework 4.0, bylo nutné upgradovat překladač také na platformu .NET Framework 4.0. Pro celý vestavný systém, který se generoval do jednoho souboru bylo vhodnější rozdělit do jednotlivých tříd a vytvořit class library s názvem `EmbeddedSystem`, která obsahuje všechno potřebné pro chod vestavného systému a slouží jako nižší vrstva pro instanci simulátoru, který generuje kompozici vestavného systému do canvasu a managuje celý vestavný systém. Další změnou je, že XML konfigurace se generuje při každém spuštění překladače.

5.1 Popis překladače

Překladač je implementován jako konzolová aplikace na platformě .NET v jazyce C#. Jako vstup pro překladač je zdrojový kód vestavného systému v jazyce e-*PFL*. Podle zdrojového kódu, který projde lexikální a syntaktickou analýzou, se vygeneruje XML soubor s konfigurací a `Class library` do cílové složky, všechny příslušné funkce a celá skladba vestavného systému do tříd v jazyce C# ve vhodné struktuře pro simulátor, který je popsán v kapitole 6. Po vygenerování vestavného systému se ještě zkopírují do cílové složky projekty, které vyžaduje simulátor a samotný simulátor. Po vygenerování potřebných projektů do cílové složky se vše přeloží překladačem C# a simulátor se spustí.

Zjednodušeně lze napsat, že překladač má jako vstup zdrojový kód vestavného systému v jazyce e-*PFL*, projekty pro simulátor v jazyku C# a samotný simulátor. Výstup překladače je vygenerovaný a spustitelný simulátor pro konkrétní vestavný systém.

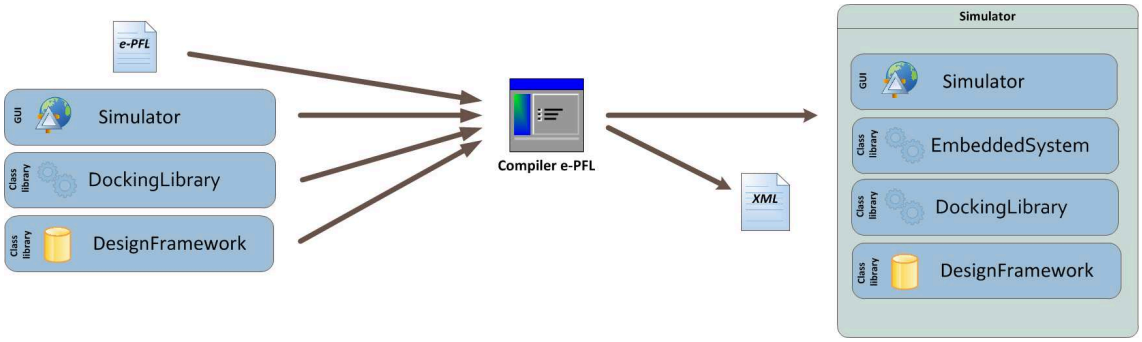

Figure 6: Schéma překladače e-*PFL*

5.2 Třída Breakpoint

Důležitou částí pro rozšíření překladače bylo implementovat debugovací funkce pro vestavný systém, které zastaví běh simulátoru, když vrátí hodnotu false. Vstupy pro vestavnou funkci musí být asociované s `EmbeddedVariables` a návratová hodnota této funkce musí být datového typu `Boolean`. Po syntaktické a sémantické analýze zdrojového kódu e-*PFL* se vygeneruje XML konfigurace tohoto vestavného systému. Debugovací funkce se definují globálně na celý vestavný systém je nutné toto pravidlo dodržet i při generování XML konfigurace.

5.2.1 Spuštění překladače s breakpointy

Pro vygenerování simulátoru s breakpointy je nutné spouštět překladač s následující syntaxí:

```
e−EPL.exe −w[test1] vending2.pfl
```

Kdy:

- *e-EPL.exe* — název překladače e-*PFL*.
- *-w* — přepínač pro volbu překladu do simulátoru na platformě WPF.
- *[test1]* — zde se definují, které breakpointy ze zdrojového souboru se mají generovat do vestavného systému. V případě kdy je potřeba více breakpointů, tak se jednotlivé breakpointy oddělují čárkou.
- *vending2.pfl* — název zdrojového souboru.

5.2.2 Generování breakpointů z e-*PFL*

Následující příklad znázorňuje generování breakpointu se jménem `test1` s podmínkou, když je vhozena mince do automatu s nominální hodnotou 50 Kč. V případě, kdy je tato podmínka splněna, simulace se zastaví v debugovacím módu simulátoru. Popis breakpoint funkce je ve výpisu 1 v kapitole 4.3. V normálním módu simulátoru jsou všechny breakpointy ignorovány.

Během překladu se provede sestavení XML konfigurace vestavného systému. Veškeré breakpointy se generují za definicí všech komponent vestavného systému do XML elementu s názvem `Breakpoints`. Toto pravidlo vychází z definice, že breakpointy jsou definovány na celý vestavný systém.

```
<Configuration project="vending2.conf.xml" created="4/13/2011 12:51 PM">
  <EmbComponent name="Simulator" mediator="Simulator">
    .
    .
    .
  </EmbComponent>
  <Breakpoints>
    <Breakpoint ID="test1">
     <Process>
       <EmbeddedProcess name="test1">
          <Input>
            <EmbeddedVariable ref="coin" />
          </Input>
          <Output />
        </EmbeddedProcess>
      </Process>
    </Breakpoint>
  </Breakpoints>
</Configuration>
```

Výpis 2: Struktura XML konfigurace vestavného systému

V XML souboru se vygeneroval jeden breakpoint s ID `test1`, který obsahuje právě jeden vestavný proces s totožným názvem `test1`. Tento vestavný proces přijímá na vstupu proměnnou `coin` podle definice z e-*PFL* a nevydává dále žádný výstup.

Po vygenerování XML souboru se vygeneruje breakpoint funkce, která se syntaxí neodlišuje od vestavných funkcí zařízení. Breakpoint obsahuje metodu Evaluate, která aplikuje potřebný vstup pro breakpoint funkci a vrací hodnotu, která je dále evaluována v simulátoru. Dle konvence vrací každá vestavná funkce pro breakpointy datový typ `Boolean`.

Každých 50ms se kontroluje stav vestavného systému a vyhodnocují se breakpointy. Před samotným vyhodnocením se zavolá `BreakpointManager` (popsán v Kapitole 5.3), který obsahuje `SortedDictionary` a postupně se evaluují breakpointy s příznakem `IsSet = true`. Breakpointy se mohou ještě v simulátoru dodatečně aktivovat či deaktivovat.

5.3 Třída BreakpointManager

Tato třída slouží jako manager pro breakpointy a definují se zde společné vlastnosti breakpointů. Třída Breakpoint je popsána v Kapitole 5.2.

5.4 Zdrojový funkcionální kód vestavné funkce

Ve vygenerované instanci simulátoru je potřebné v debugovacím režimu zobrazovat funkcionální kód vestavné funkce. Proto je nutné modifikovat `EmbeddedSystem` o rozšíření konstruktoru abstraktní třídy `Function` z původní struktury

```
protected Function(Function parent)
```

na následující strukturu

```
protected Function(Function parent, string description)
```

V abstraktní třídě `Function` se do konstruktoru přidá parametr datového typu `string` s názvem `description`, který slouží pro uchování funkcionálního kódu vestavné funkce.

Pro generování konkrétních vestavných funkcí, které dědí z abstraktní třídy `Function` je nutné doimplementovat kód abstraktní třídy `CSharpUsingGenerator` pro generování statického stringu s názvem `Description` do které se generuje její funkcionální kód. Syntaxe `f.Expression.ToString()` je funkcionálním kódem vestavné funkce.

```
.
.
sbcl.AppendLine("\tclass " + createFunctionName(f) + " : Function");
sbcl.AppendLine("\t{");
sbcl.AppendLine("\t\tpublic static string Description = \"" + f.Expression.ToString().Replace("\n"
    , "_newLine_") + "\";");
sbcl.AppendLine("\t\tpublic " + createFunctionName(f) + "(Function parent, string description):
    base(parent, description) { paramsCount = " + f.GetParameterCount() + "; }");
.
.
```

V případě, že se jedná o primitivní funkci, má tato proměnná vygenerovanou hodnotu `Description = ""`.

5.5 Referenční funkce k vestavné funkci

Dalším rozšířením implementace kompilátoru jsou referenční funkce k vestavné funkci. Tzn. funkce, které daná funkce volá a používá. Z tohoto důvodu je nutné ještě dále rozšířit řešení abstraktní třídy `Function` z Kapitoly 5.4 na strukturu

```
protected Function(Function parent, string description, string references)
```

A pro zachování hierarchie volání se všechny funkce vygenerované překladačem vkládají do statického listu s názvem `CallFunctions`. Do konstruktoru se přidá další parametr datového typu `string` s názvem `references`.

Do metody s názvem `generateExpression`, která se nachází v abstraktní třídě `CSharpUsingGenerator` se přidá parametr typu `string` s názvem `functionName`. Tento parametr obsahuje název vygenerované funkce a slouží pro zachycení hierarchie volání. Ve větvi `if (e is Identifier)` se přidá následující kód, který vkládá do listu `CallFunctions` objekt `CallFunction`, který obsahuje dvě proměnné. Do první proměnné s názvem `ParentFunction` se ukládá rodičovská funkce a do druhé proměnné s názvem `ChildrenFunction` se ukládá její potomek. Obě proměnné jsou datového typu `string`. Tímto postupem se naplní list všemi vygenerovanými funkcemi.

```
GeneratorWPF.RuntimeGenerator.CallFunctions.Add(new CallFunction { ParentFunction =
    functionName, ChildrenFunction = f.Name });
```

Před generováním konkrétních vestavných funkcí, které dědí z abstraktní třídy `Function` se zavolá privátní metoda s názvem `GetReferences`, která prochází již naplněný list `CallFunctions` a sestaví `string` s referenčními funkcemi. Metoda má jeden parametr datového typu `string` podle kterého se prochází list a vrací se hodnota typu `string`.

Ještě bylo nutné doimplementovat kód abstraktní třídy `CSharpUsingGenerator` pro generování statického stringu s názvem `References` do které se generují referenční funkce, které vrací privátní metoda `GetReferences`. Jednotlivé vestavné funkce se od sebe oddělují pouze čárkou.

```
.
.
string  references = GetReferences("");
int  paramCount = f.GetParameterCount();
sbcl.AppendLine("\tclass " + createFunctionName(f) + " : Function");
sbcl.AppendLine("\t{");
sbcl.AppendLine("\t\tpublic static string Description = \"\";");
sbcl.AppendLine("\t\tpublic static string References = \"" + references + "\";");
sbcl.AppendLine("\t\tpublic " + createFunctionName(f) + "(Function parent, string description,
    string references):base(parent, description, references) { paramsCount = " + paramCount
    + "; }");
.
.
```

V případě, že se jedná o primitivní funkci, má tato proměnná vygenerovanou hodnotu `References = ""`, protože primitivní funkce dále nepoužívají žádné funkce.

5.6 Proces generování instance simulátoru

Jak již bylo nastíněno v Kapitole 5.2.2 proces generování instance simulátoru se generuje ze zdrojového kódu e-*PFL*. Kapitola ovšem nepopisuje celý proces generování. Proto je nutné definovat celý proces vygenerování instance simulátoru počínaje spuštěním překladače s potřebnými parametry a konče spuštěním výsledného simulátoru na platformě WPF.

Celý proces se dá shrnout do následujících kroků:

1. Nejprve je nutné spustit překladač se syntaxí, která je popsána v Kapitole 5.2.1. Překladač je možné spouštět také i bez breakpointů.
2. Načte se zdrojový kód e-*PFL* a provede se lexikální a syntaktická analýza kódu.
3. Vytvoří se složka s názvem `Simulator` a do ní se zkopírují projekty, které jsou obsažené v solutioně pro simulátor. Class library `EmbeddedSystem` obsahuje kostru pro vestavný systém ve které jsou rozděleny třídy z původních souborů `SimulatorMainBasic`, `SimulatorPrimitives` a `SimulatorStandard`.
4. Vygeneruje se XML konfigurace vestavného systému a do této konfigurace se přidají breakpointy do XML elementu `Breakpoints` za XML element `EmbComponent`.
5. Vygenerovaná XML konfigurace se použije pro sestavení komponent, zařízení, vygenerování vestavných a breakpoint funkcí. Tyto vestavné a breakpoint funkce se generují do složky `EmbeddedFunction` v class library `EmbeddedSystem`. Při generování vestavných funkcí se u každé vygenerované funkce musí upravit soubor s názvem `EmbeddedSystem.csproj`, který obsahuje XML definici C# projektu. Do této definice je nutné Includovat vygenerované funkce (třídy). Jinak není možné solutionu automaticky zkompilovat a spustit.
6. Do veřejné statické metody `void GetStartConfiguration()` ve třídě `Settings.cs`, která je v kořenové složce class library `EmbeddedSystem`, se vygenerují SortedDictionary `embeddedVariables`, `generatedFunctions` a celá kompozice vestavného systému.

7. Celá solutiona se zkompiluje kompilátorem pro C# verze 4.0.30319. Informace o kompilaci se uloží do logu s názvem `msbuild.log`, který se nachází v adresáři `Simulator`.

8. Po úspěšném zkompilování se simulátor automaticky spustí.

Pro informaci se celý tento proces uskuteční průměrně za 19 vteřin na sestavě AMD Turion X2 RM-70 (2000 MHz), 3836 MB (DDR2-800 Registered DDR2 SDRAM), HDD 320 GB (5400 RPM, SATA-II) s OS Windows 7.

6 Simulátor e-$\mathcal{PFL}$

Pro simulování vestavných systému bylo nutné implementovat aplikaci, ve které bude kompilátorem vygenerovaný vestavný systém zobrazován a uživatel může interagovat a ladit vestavné systémy při neočekávaném chování. Ladění funkcionálního kódu, který je uvnitř jednotlivých procesů se tato práce vůbec nezabývá. Simulátor se zabývá koordinací a komunikací jednotlivých jednotek a procesů, ze kterých se vestavné systému skládají.

V této kapitole je dále popsána implementace vícevrstvé aplikace pro řízení simulace vestavných systémů vygenerované kompilátorem e-$\mathcal{PFL}$ na platformě WPF 4.0. Obecně je tato aplikace jen jako uživatelské rozhraní, která zobrazuje vestavný systém a obsahuje funkce, kterými je vestavný systém řízen a debugován. Samotný vestavný systém je vygenerován překladačem e-$\mathcal{PFL}$ jako samostatná `Class library`, do které simulátor pouze přistupuje.

6.1 Zadání

Pro současné řešení je nutné realizovat sofistikovanější nástroj pro simulaci průběhu vykonávání a ladění vestavných systémů. Nástroj by měl zobrazovat celou strukturu vestavného systému a měl by obsahovat interakci s tímto vestavným systémem.

6.1.1 Funkční požadavky

- Simulátor bude realizován jako desktopová aplikace na platformě WPF s dynamicky generovaným obsahem (závislým na zvolené konfiguraci vestavného systému).
- Základní layout rozložení částí vestavného systému bude do n-úhelníku (n = dle počtu prvků) s funkcí drag-and-drop pro rozmístění těchto částí do canvasu pro možnost uložení tohoto rozložení.

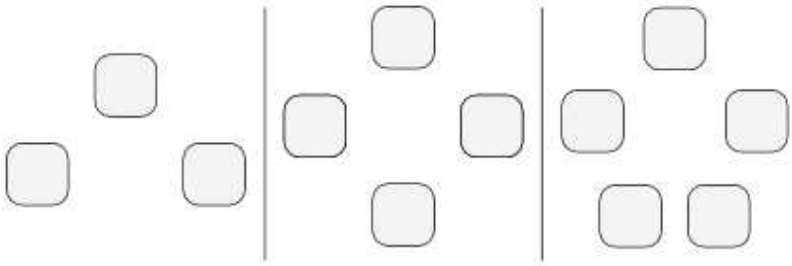

Figure 7: Rozložení částí vestavného systému (3, 4 a 5 zařízení)

- Bude obsahovat debugovací mód pro krokování chování jednotlivých zařízení.
  - jemné krokování (po jednotlivých stavech zařízení)

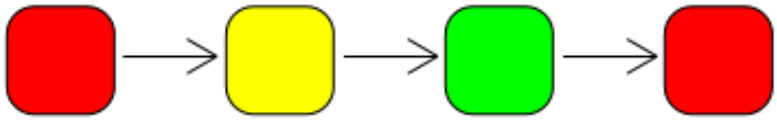

Figure 8: Jemné krokování zařízení

  - hrubé krokování (vždy na inicializační stav zařízení)

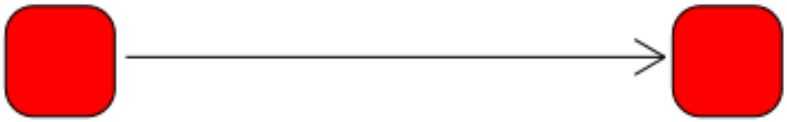

Figure 9: Hrubé krokování zařízení

- Pro každé zařízení bude detail příslušné komponenty, ke které patří. Její vestavné funkce se zdrojovým funkcionálním kódem a další vestavné funkce, která daná funkce volá s možností výběru vestavné funkce. Po výběru funkce se zobrazí její zdrojový funkcionální kód a další vestavné funkce, která vybraná funkce volá.
- Ovládání nastavení frekvence celého systému nebo jednotlivých zařízení. Pro ovládání frekvence celého systému bude posuvník na viditelném místě. Nastavování jednotlivých zařízení bude v externím okně.
- Ovládání nastavení breakpointů pro možnost debugování (aktivovat/deaktivovat jednotlivé breakpointy). Nastavování breakpointů bude v externím okně.

6.1.2 Nefunkční požadavky

- Aplikace by měla být přehledná a jednoduchá na ovládání.
- Zbytečně hardwarově nenáročná.

6.2 Architektura aplikace

Architektura je podle [18] všeobecné označení určující celkovou strukturu a základní konstrukci částí nebo kompletního počítačového systému. V našem speciálním případě architektury aplikace se jedná o způsob rozdělení aplikace, aplikačních dat, procesů i datových toků do logických celků, stanovení struktury těchto komponent, vzájemných vztahů a interakcí mezi nimi, a to na dostatečně obecné úrovni. Je zřejmé, že se k návrhu a volbě architektury musí přistupovat na samém počátku vývoje jakékoliv aplikace, a to s velkou opatrností a rozmyslem. Jakékoliv její pozdější změny jsou totiž velmi komplikované nebo téměř nemožné.

Stávající řešení simulátoru bude realizováno do dvou vrstev z důvodu, že Embedded Systém mění rychle své vnitřní stavy komponent a business vrstva by jen zbytečně zabírala režii ve vykonávání a celkové zátěži na hardware. Každých 50ms se kontroluje a generuje stav všech komponent a komunikačních kanálů Embedded Systému.

Vrchní vrstva (Simulator) slouží jako prezentační pro zobrazení Embedded Systému a spodní vrstva (EmbeddedSystem) slouží jako datová vrstva s konfigurací celého vestavného systému. Prezentační vrstva dále používá pro zobrazování dynamicky generovaného obsahu DesignFramework ve kterém jsou definovány základní tvary pro komponenty Device a komunikačních kanálů. Dále je použita User Controla s názvem DockingLibrary pro dokovací panely v prezentační vrstvě.[2]

Pro možnost rozšíření řešení na třívrstvou architekturu je možné přidat business vrstvu mezi Simulátor a Embedded Systém. Funkce této business vrstvy je zajištění poskytování objektů pro Simulátor. Zejména komponenty Device, příslušných vestavných funkcí těchto komponent a breakpointy pro zastavování chodu simulace a následného ladění celého vestavného systému. Tato třívrstvá architektura je vhodná v případě, kdy je nutné měnit zobrazovací vrstvu za jinou nebo má sloužit tato business vrstva pro více druhů prezentačních vrstev (webové stránky, webové služby, RIA).

Prezentační vrstva je dvouvláknová a je použita pomocná třída BackgroundWorker pro aktualizaci UI prvků aplikace. Použití více vláken je z důvodu řízení simulace. První vlákno se stará o vykonávání simulace vestavného systému a druhé vlákno se stará o samotné řízení simulace. Kdyby bylo použito pouze jediné vlákno, tak se simulace spustí a simulátor se nedá ovládat.

[2]Tato kontrola není standardně ve Visual Studiu 2010 implementována.

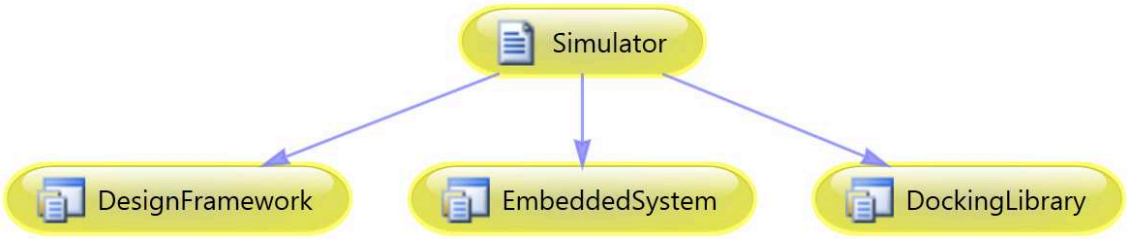

Figure 10: Architektura simulátoru

6.3 BackgroundWorker

V raném návrhu WPF ustanovili tvůrci nový threading model s názvem *called thread rental* umožňující přistupovat k UI z jakéhokoliv vlákna pro snížení režie zamykání a skupiny souvisejících objektů by byly seskupeny pod jediný zámek (tzv. kontext). Bohužel by tento návrh přinesl další složitost pro jedno-vláknové aplikace a bylo by více obtížnější spolupracovat se staršími kódy (Win32 API). Nakonec byl tento plán zamítnut.

Výsledkem je, že WPF podporuje *single-threaded apartment*, který je podobný modelu, který se používá ve Windows Forms aplikacích.

Má několik základních pravidel:

- WPF UI prvky mají vláknovou afinitu. Vlákno, které vytvořilo prvek je vlastníkem tohoto prvku a ostatní vlákna nemůžou přímo přistupovat k tomuto prvku.
- WPF objekty mají vláknovou afinitu odvozenou z DispatcherObject. DispatcherObject ověřuje jestli je kód pro UI spouštěn ve správném vlákně.
- Jedno vlákno běží celou aplikaci a vlastní všechny WPF objekty.

V případě porušení pravidla (přístup k UI prvku z jiného vlákna) se vyvolá výjimka InvalidOperationException se zprávou *The calling thread cannot access this object because a different thread owns it*.

Pro použití více vláken, které přistupují k UI prvkům je nutné implementovat BackgroundWorker. BackgroundWorker je pomocná třída v System.ComponentModel pro správu pracovních vláken.

Poskytuje následující věci:

- Vlastnost „Cancel“ pro signalizování vláknu, aby skončilo i bez volání metody Abort.

- Protokol pro podávání zpráv o průběhu práce, dokončení a zrušení práce.
- Implementace rozhraní IComponent, které BackgroundWorkeru dovoluje, aby se s ním dalo pracovat ve Visual Studio Designeru.
- Zachycování výjimek v pracovním vláknu.
- Schopnost aktualizovat WPF ovládací prvky na základně průběhu vlákna.

Poslední dva body jsou pro tuto implementaci nejužitečnější – nemusí se ve spouštěných metodách pomocí BW používat try/catch bloky a UI prvky se aktualizují i bez volání Control.Invoke. BW využívá tzv. fond vláken (thread-pool), který spravuje vytvořená vlákna a sám ukončuje jejich práci. Z tohoto důvodu se nevolá na BackgroundWorker vlákno metoda Abort, o to se sám postará fond vláken.

Pro užití BackgroundWorker v aplikaci bylo nutné:

- Vytvořit instanci třídy BackgroundWorker a vytvořit handler pro událost DoWork. Událost DoWork spouští všechny zařízení vestavného systému a obsahuje nekonečnou smyčku pro vykonávání simulace.

```
private void backgroundWorker_DoWork(object sender, DoWorkEventArgs e)
{
    foreach (var device in DeviceManager.Devices)
    {
        device.Value.Start();
    }
    while (true)
    {
        Worker.StartProcess(backgroundWorker);
        if (backgroundWorker.CancellationPending)
        {
            e.Cancel = true;
            return;
        }
        e.Result = 1;
    }
}
```

Výpis 3: Handler pro událost DoWork

- Zavolat metodu RunWorkerAsync (nastartuje instanci BW).

Jako argument metody RunWorkerAsync se může zadat cokoliv, přijímá totiž typ object. Zadaný argument se následně předá zpracovateli události DoWork. Kromě DoWork se v BackgroundWorkeru nachází událost RunWorkerCompleted, která se zavolá, když DoWork dokončí své vykonávání. Zpracování této události není povinné, ovšem většinou je to užitečné, například pro zpracování výjimek, které vzniknou během DoWork. Zpracovatel této události může přímo přistupovat k WPF UI prvkům bez explicitního marshallingu, zatímco DoWork nemůže.

Pro ohlašování pokroku práce bylo nutné udělat následující:

- Nastavit vlastnost WorkerReportsProgress na true.
- V DoWork handleru volat metodu ReportProgress, která informuje o průběhu vykonávání každých 50 milisekund.

```
public static void StartProcess(BackgroundWorker backgroundWorker)
{
    if (backgroundWorker.WorkerReportsProgress)
    {
        lock (backgroundWorker)
        {
            Thread.Sleep(50);
            backgroundWorker.ReportProgress(1);
        }
    }
}
```

Výpis 4: Metoda StartProcess

- Vytvořit handler události ProgressChanged, který se dotazuje na průběh vykonávání. Kód uvnitř ProgressChanged zpracovatele může, stejně jako RunWorkerCompleted, volně komunikovat s UI prvky aplikace. A právě toto je místo, které se využívá k aktualizování UI prvků aplikace. Aktuálního stav vestavného systému se zobrazí v aplikaci (hodnoty komunikačních kanálů, stavy zařízení, vestavné funkce zařízení atd.).

Pro ukončení práce vlákna bylo nutné:

- Nastavit vlastnost WorkerSupportsCancellation na true.

- V DoWork handleru sledovat stav vlastnosti CancellationPending. Pokud je její hodnota true, nastavit argument „e“ u DoWork na Cancel = true;.
- Zavolat CancelAsync pro ukončení práce, která automaticky zavolá událost RunWorkerCompleted. Metoda která se stará o událost RunWorkerCompleted zastavuje všechny zařízení vestavného systému a simulace se ukončí.

```
private void backgroundWorker_RunWorkerCompleted(object sender,
     RunWorkerCompletedEventArgs e) {
    if (e.Cancelled)
    {
        foreach (var device in DeviceManager.Devices)
        {
            device.Value.Abort();
        }
    }
    else if (e.Error != null)
    {
        // An error was thrown by the DoWork event handler.
        MessageBox.Show(e.Error.Message, "An␣Error␣Occurred");
    }
    StopSimulation();
}
```

Výpis 5: Handler pro událost RunWorkerCompleted

6.4 Dynamická analýza

Simulátor je implementován univerzálně pro jakýkoliv vestavný systém. V následující části podkapitoly je popsáno generování vestavného systému do canvasu. Podle konfigurace vestavného systému se vygeneruje konkrétní kompozice vestavného systému.

6.4.1 Generování vestavného systému do canvasu

Generování kompozice vestavného systému je možné shrnout do následujících kroků:

1. Při spouštění aplikace se konstruktoru `MainWindow` zavolá veřejná statická metoda `void GetStartConfiguration()` ve třídě `Settings`, která naplní SortedDictionary `embeddedVariables`,`generatedFunctions`,vytvoří všechny zařízení, jejich vestavné funkce a breakpoint funkce vestavného systému.

2. V konstruktoru se dále zavolá se privátní metoda void `GenerateContent()`, která generuje kompozici do canvasu. Algoritmus pro generování se skládá z těchto částí:

   (a) Vykreslení zařízení do canvasu, které jsou definovány ze zdrojového kódu e-*PFL*. V průběhu postupného vykreslování těchto zařízení se procházejí všechny vestavné procesy zařízení a jejich vstupy a výstupy se přidávají do privátního listu `List<Input> inputs` resp. `List<Output> outputs`, který obsahuje objekt `Input` resp. `Output`, který má dvě vlastnosti `Variable` a `Device`. Obě vlastnosti jsou datového typu `string`.

   (b) Provede se analýza vstupů a výstupů podle které se určí a doplní vstupní zařízení. Vstupní zařízení a analýza je popsána v Kapitole 6.7.

   (c) Po doplnění vstupních zařízení se všechny zařízení v canvasu rozmístí do n-úhelníku.

   (d) Do canvasu se vykreslí všechny komunikační kanály mezi příslušnými zařízeními.

3. Aplikace se nastaví do stavu pro normální mód.

6.5 Stavová analýza

Zařízení ve vestavném systému se v průběhu vykonávání procesu dostavájí do různých stavů. Každý stav zařízení je indikován v simulátoru jinačí barvou.

6.5.1 Zařízení

Vestavný systém se skládá ze zařízení (datový typ Device), které obsahují vestavné procesy. Tyto vestavné procesy požadují pro své vykonávání vstupy a produkují požadovaný výstup. Zařízení má procesy uložené ve stromové struktuře, ze které se postupně vybírají procesy pro vykonávání. Pokud má zařízení `TreeNode` typu `FairNode`, po vykonání procesu se přesune na konec stromu a je na řadě další vestavný proces. Struktura `TreeNode` je popsána v Kapitole 6.6.1. Vzájemná komunikace mezi zařízeními je uskutečněná komunikačním kanálem. Tento komunikační kanál může spojovat dvě různé zařízení nebo samo sebe. Každý komunikační kanál má pouze jeden vstup a pouze jeden výstup.

Zařízení se v průběhu vykonávání vybraného procesu dostane do těchto stavů:

- **Nečinný** — zařízení nemá vybraný žádný proces pro vykonání. Zařízení může obsahovat i více vestavných procesů. V simulátoru se tento stav indikuje červeným pozadím zařízení.
- **Čekající** — zařízení vybralo proces a aplikují se potřebné proměnné na vstupy procesu pro následné vykonání. Při výběru musí být nejdříve splněna podmínka, že všechny vstupní komunikační kanály vestavného procesu obsahují hodnotu. V jedné chvíli může být vykonáván maximálně jeden proces. V simulátoru se tento stav indikuje žlutým pozadím zařízení.
- **Vykonávající** — vybraný proces se vykonává a postupně se konzumují proměnné dalšími zařízeními, které čekají na své vstupy. Hodnotu lze do komunikačního kanálu zapsat ve chvíli, kdy je komunikační kanál volný a neobsahuje žádnou hodnotu. Další proces nemůže být vybrán dokud se celý proces neprovede a všechny hodnoty jsou zkonzumovány. V simulátoru se tento stav indikuje zelenám pozadím zařízení.

Figure 11: Stavy zařízení

Implementace zařízení je v příloze A, Výpis 9.

6.6 Statická analýza

Následující podkapitola vysvětluje objekt TreeNode. TreeNode a její potomci se ve vygenerovaném vestavném systému hojně používají.

6.6.1 TreeNode

Základní (bázovou) třídou je abstraktní třída `TreeNode`, ze které dědí abstraktní třída `BranchNode` a konkrétní třída `ProcessNode`. Z abstraktní třídy `BranchNode` dále dědí konkrétní třídy `FairNode` a `UnfairNode`.

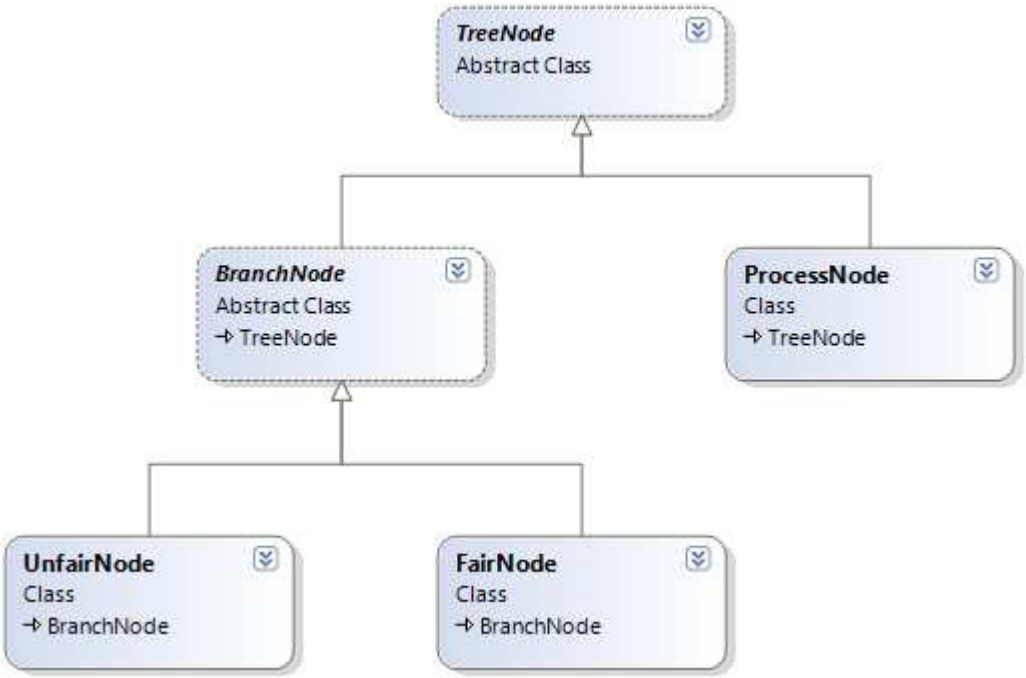

Figure 12: Hierarchická struktura dědění z abstraktní třídy `TreeNode`

Třída `FairNode` nebo `UnfairNode` se používá pro tvorbu definice stromu funkcí vestavného zařízení. Třída `ProcessNode` prezentuje vestavnou funkci má definované pole komunikačních kanálů pro vstupy a pole komunikačních kanálů jako výstupy.

Použití:

```
TreeNode tree_Dev2 = new FairNode(new List<TreeNode>{new ProcessNode("serve", new
    serve_11281761(null, serve_11281761.Description, serve_11281761.References), "
    serve_11281761",
  new EmbeddedVariable[]
```

```
        {
            embeddedVariables["choice"]
        },
        new EmbeddedVariable[]
        {
            embeddedVariables["servedItem"],
            embeddedVariables["initCounter"],
            embeddedVariables["displayInput2"]
        }),
        new ProcessNode("paid", new paid_59899045(null, paid_59899045.Description,
              paid_59899045.References), "paid_59899045",
        new EmbeddedVariable[]
        {
            embeddedVariables["servedItem"],
            embeddedVariables["paid"]
        },
        new EmbeddedVariable[]
        {
            embeddedVariables["soldItem"],
            embeddedVariables["displayInput2"]
        })
    });
    Device Dev2 = new Device(tree_Dev2, verbrose, "Dev2");
```

Výpis 6: Definice stromu vestavných funkcí zařízení

6.7 Vstupní zařízení pro vestavný systém

Pro vestavný systém jsou vstupy nedílnou a nutnou součástí pro provoz celého systému. Ve zdrojovém kódu e-*PFL* se vstupní zařízení přímo neuvádí, jen jako komunikační kanál se vstupní hodnotou vyprodukovanou odnikud. Proto se před generování kompozice vestavného systému do canvasu provede analýza vstupů a výstupů jednotlivých zařízení. Všechny vstupy a výstupy se spárují a v případě, kdy se některé vstupy a výstupy nespárují, identifikují se takto místa pro vstupní zařízení do vestavného systému. Po zadání vstupní hodnoty uživatelem se tato hodnota zapíše do komunikačního kanálu a hodnota může být dále konzumována zařízením, které má tuto hodnotu jako vstup.

Vstupní zařízení se značí šedou barvou pozadí a s název se skládá z prefixu „In" a názvu vestavné proměnné komunikačního kanálu v kapitálce, do kterého proměnná vstupuje. Vstupní zařízení přijímá pouze vstupní hodnoty datového typu `Integer`, `Char` a `Bool`. Dialog pro vstupní zařízení je popsán v Kapitole 6.8.4.

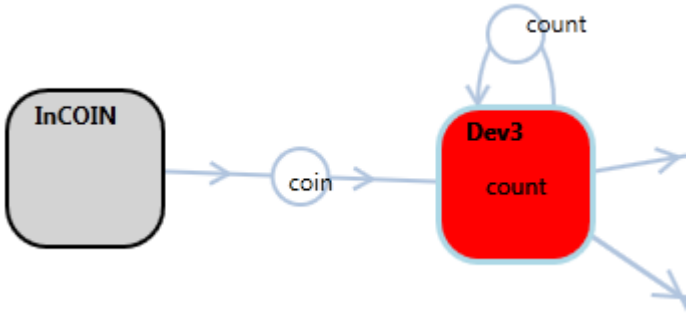

Figure 13: Vstupní zařízení InCoin pro vestavný systém

6.8 Návrh uživatelského rozhraní

Uživatelské rozhraní aplikace představuje hlavní komunikační prostředek mezi aplikací a jeho uživateli. Návrhu uživatelského rozhraní by proto měla být věnována velká pozornost. Na webu Usability Post lze najít článek o základních pravidlech tvorby a fungování uživatelského rozhraní.

Jedná se o těchto osm zásad, kdy každou reprezentuje jedna vlastnost. Uživatelské rozhraní by tedy mělo být:

- **Srozumitelné** – Mělo by uživateli poskytovat dodatečné informace, co která akce způsobí. Je však nutné dát si pozor na přehlcení nápovědou.
- **Stručné** – Pokud lze použít méně a jednodušších prvků, je vhodné to udělat. Určité akce lze znázornit grafiky mnohem srozumitelněji, než pomocí mnoha slov.
- **Povědomé** – Uživatelé mají obvykle zafixovány scénáře ovládání různých typů aplikací. Je tedy dobré respektovat zažité principy namísto experimentování.
- **Reagující** – Uživatelské rozhraní by mělo reagovat na každou akci uživatele. Pokud uživatel klikne na tlačítko, měl by vidět efekt zmáčknutí onoho tlačítka. Zároveň by rozhraní mělo být rychlé.
- **Soudržné** – Jednotlivé aplikace stejné řady nebo od stejné firmy využívají podobné uživatelské rozhraní. To umožňuje uživatelům předvídat, jak se bude chovat určitý nástroj v jiné aplikaci stejné řady.
- **Atraktivní** – Uživatelské rozhraní by mělo být příjemné a neobtěžovat. Pokud bude rozhraní ošklivé, je pravděpodobné, že běžný uživatel bude hledat jiné a hezčí.

- **Účinné** – důležitý je zde odhad uživatele a co bude chtít udělat. Upravení rozhraní může uživateli nabídnout účinnou cestu k dosažení požadovaného výsledku rychleji.
- **Odpouštějící** – Každý známe situace, kdy občas uděláme něco, co jsme nechtěli a musíme se vrátit. Uživatelské rozhraní by mělo nabídnout možnost vrátit zpět chybné kroky.

6.8.1 RibbonMenu

Menu simulátoru je ve struktuře Ribbonu (česky označovaný jako „pás karet") , které bylo poprvé představeno v kancelářském balíku Microsoft Office 2007 a po jeho příznivém přijetí byl přidán i do dalších aplikací od Microsoftu – ve Windows 7 jsou tak ribbonem vybaveny už i WordPad, Malování či Windows Live Movie Maker. Ribbon je panel obsahující tlačítka a ikony, který je umístěný v horní části okna aplikace. Panel obsahuje veškeré funkce, které program poskytuje, uspořádané do záložek. Ribbon je patentovaný Microsoftem jako ovládací prvek uživatelského rozhraní operačního systému Windows. Ve WPF 4 je tato komponenta volně dostupná.

RibbonMenu se skládá z pěti záložek:

- **Home —** zde se nachází základní ovládání simulátoru jako jsou načtení rozložení kompozice vestavného systému či ukončení aplikace.
- **Project —** tato záložka je zatím volná. Později je možné implementovat funkce, které se vztahují k projektu (nastavení velikosti a vlastnosti zařízení v canvasu, zobrazování dodatečných informací k projektu, ...).
- **Settings —** zde se nachází funkční nastavení zařízení a breakpointů. Dialog pro nastavení zařízení je popsán v Kapitole 6.8.5 a dialog pro nastavení breakpointů je popsán v Kapitole 6.8.6.
- **Debug —** zde se nachází ovládací prvky pro řízení simulace. Tlačítkem `Run` se simulace spustí v normálním režimu, který je popsán v Kapitole 6.8.2, tlačítkem `Debug` se simulace spustí v debugovacím režimu, který je popsán v Kapitole 6.8.3. Simulace se v debugovacím režimu dá pozastavit tlačítkem `Breakpoint` s následnou možností hrubého krokování tlačítkem `Step Over` nebo jemným krokováním tlačítkem `Step Into` a simulace se dá vypnout kdykoliv během vykonávání tlačítkem `Stop`. Ukázka záložky v Debug je na obrázku 14.

- **Help —** zde se nachází informace o aplikaci. Později je možné implementovat nápovědu, jak se s touto aplikací zachází.

Figure 14: RibbonMenu

6.8.2 Normální mód

Normální mód je odlehčený model pro simulaci, protože se zde každých 50ms nekontrolují a neevaluují breakpoint funkce. Celý canvas je roztáhnutý na 100% šířky a délky aplikace. V levé horní části canvasu je název zdrojového souboru e-*PFL* bez přípony a aktuální datum s časem, kdy byl spuštěn překlad zdrojového souboru e-*PFL*. Na levé části canvasu se nachází zásobník s hodnotami z `embeddedVariables` ve tvaru „název proměnné = hodnota proměnné". Na spodní části canvasu se nachází horizontální posuvník pro změnu nastavení globální rychlosti simulace vestavného systému s ukazatelem aktuální rychlosti. Canvas umožňuje techniku drag-and-drop, která je popsaná v Kapitole 6.9.

Tento mód pro simulaci nebude zřejmě až tak často používaný.

Obrázek aplikace v normálním módu se nachází na následující stránce.

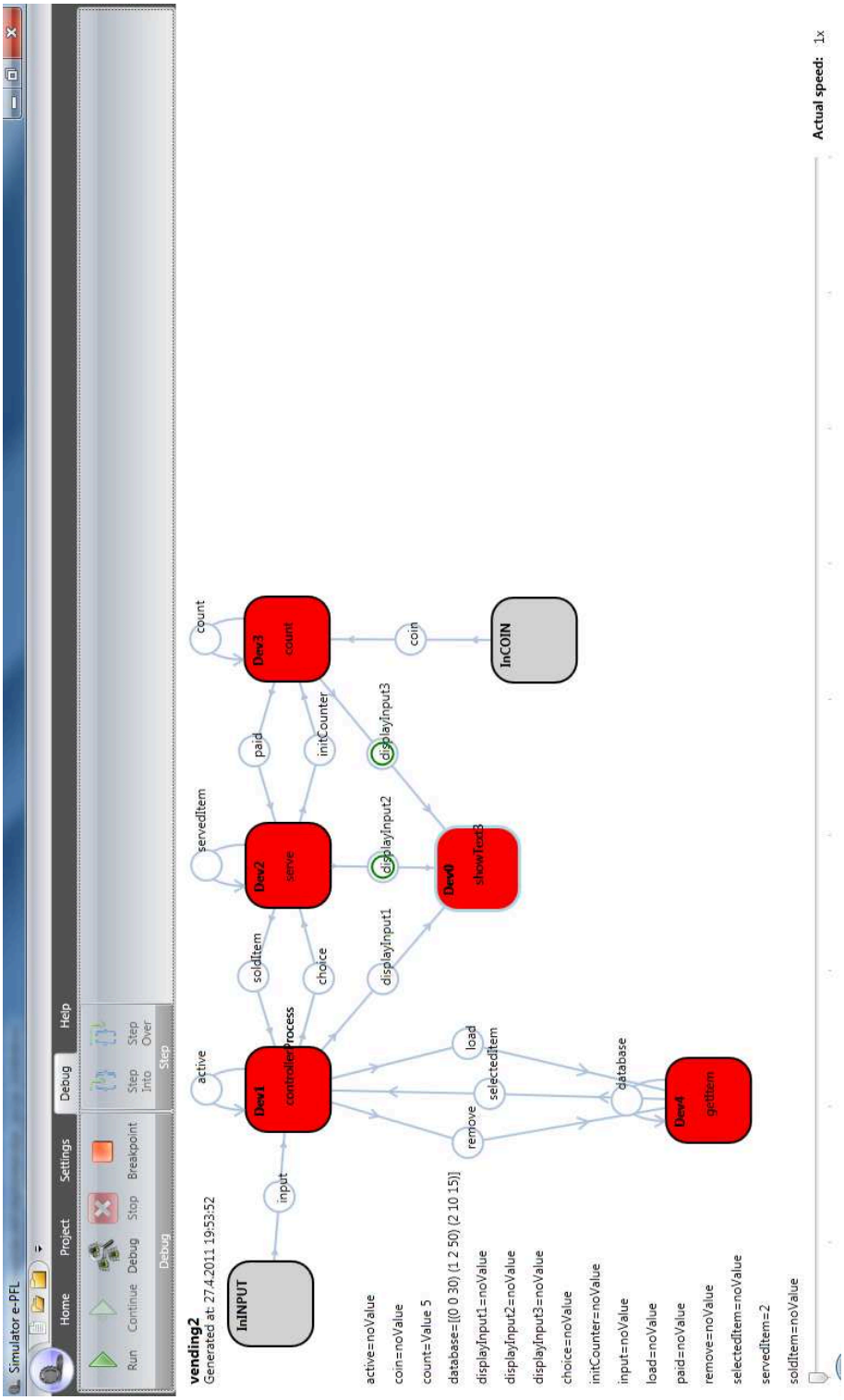

Figure 15: Normální mód simulace

6.8.3 Debugovací mód

Debugovací mód simulace rozšiřuje normální mód a aktivuje evaluaci breakpoint funkcí. Aplikace se nyní vodorovně dělí na dvě části v poměru 11:7.

- V levé první části s názvem `View` se nachází vše co je obsažené v normálním módu. Canvas má výšku 100% aplikace a šířku zhruba 65% aplikace.
- V pravé druhé části s názvem `Code` se nachází prostor s výškou 100% a se šířkou 35% aplikace pro debugovací informace a výpisy vybraného zařízení z canvasu. Nachází se zde plovoucí okno s názvem `Tree dev`, které zobrazuje aktuální strom procesů, vstupy a výstupy procesů vybraného zařízení. Dále se zobrazují informace ve které komponentě se vybrané zařízení nachází, vestavné funkce zařízení, referenční funkce vestavné funkce a zdrojový funkcionální kód vestavné funkce. Po kliknutí na referenční funkci se zobrazí její referenční funkce a zdrojový funkcionální kód. Všechny funkce, které si uživatel prohlíží se ukládají do zásobníku a pro navigaci lze použít šipky  a  obdobně jako v internetových prohlížečích.

Canvas umožňuje techniku drag-and-drop, která je popsaná v Kapitole 6.9.

Obrázek aplikace v debugovacím módu se nachází na následující stránce.

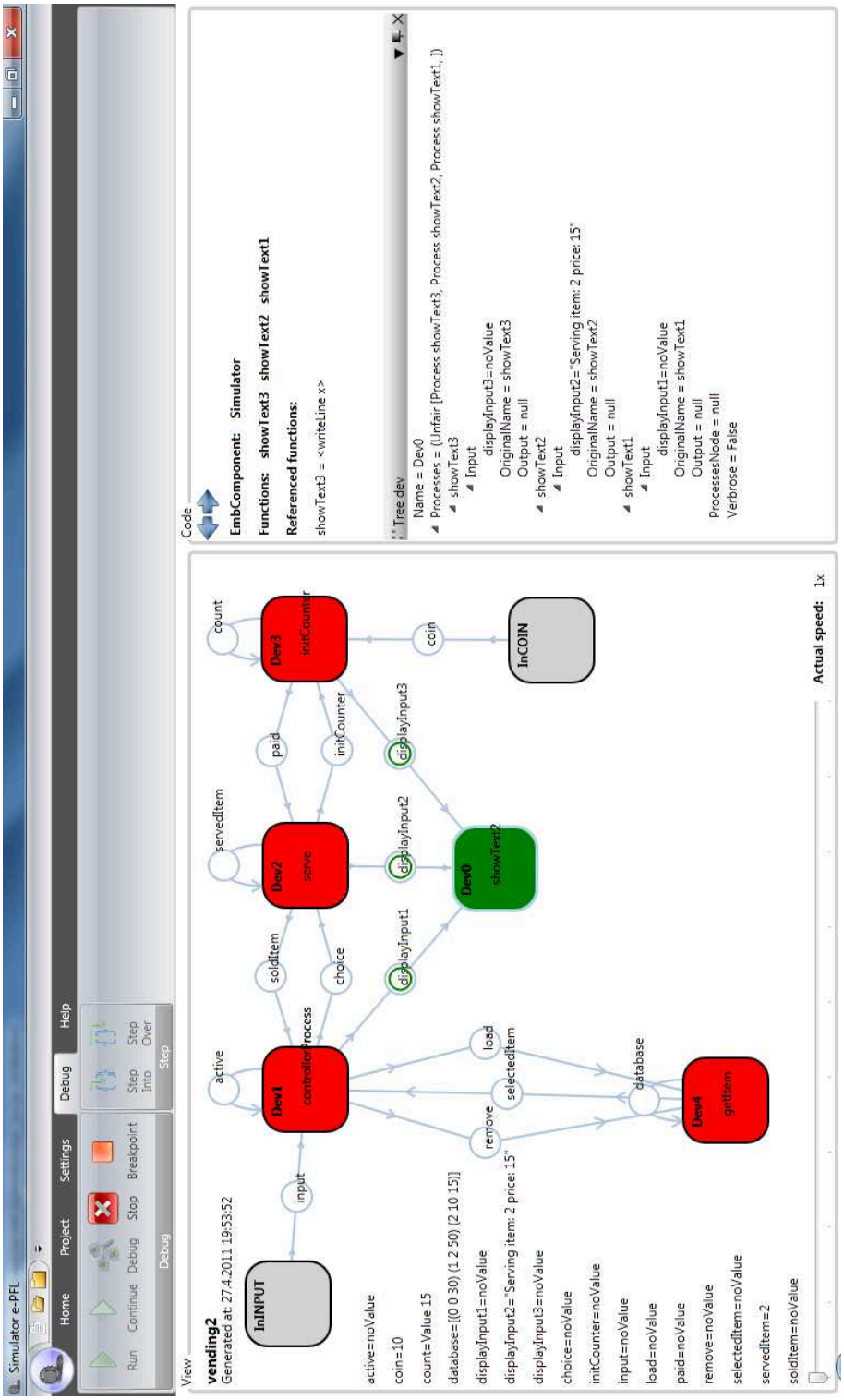

Figure 16: Debugovací mód simulace

6.8.4 Vstup pro vstupní zařízení

Dialogové okno se zobrazí po dvojkliku na vstupní zařízení, které je detailněji popsáno v Kapitole 6.7. Tlačítkem `OK` se vstup přiřadí do příslušného komunikačního kanálu a dialogové okno se zavře a simulace automaticky pokračuje dále. Tlačítkem `Storno` se celý dialog zavře a simulace je nadále zastavena.

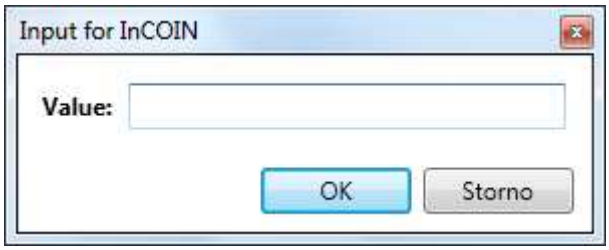

Figure 17: Dialog pro vstupní zařízení

6.8.5 Nastavení zařízení

Dialogové okno se vyvolá z RibbonMenu a obsahuje záložky, které se generují podle počtu zařízení ve vestavném systému. Zatím je možné pouze nastavovat rychlost jednoho cyklu zařízení v milisekundách. Změna se provádí okamžitě po změně hodnoty.

Figure 18: Dialog pro nastavení zařízení

Nastavení vstupního zařízení se negeneruje, protože zatím nemá žádný účel.

6.8.6 Nastavení breakpointů

Dialogové okno se vyvolá z RibbonMenu a obsahuje záložky, které se generují podle počtu breakpointů ve vestavném systému. Každý breakpoint obsahuje výpis vlastního funkcionálního kódu a možnost aktivace/deaktivace příslušného breakpointu. Při deaktivaci se breakpoint ignoruje při kontrole splnění podmínek v debugovacím režimu simulace.

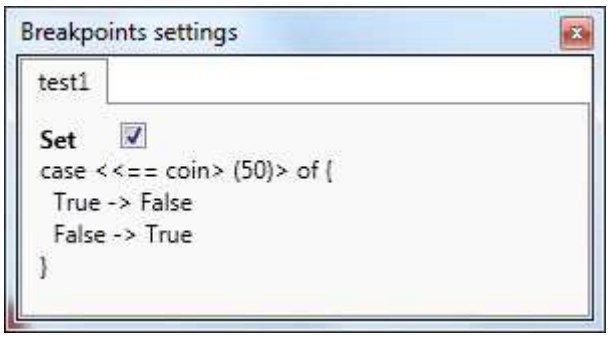

Figure 19: Dialog pro nastavení breakpointů

6.9 Technika drag-and-drop

Technika drag-and-drop (česky „táhni-a-pusť“ nebo „přetahování“) patří k užitečným prvkům, které mohou zvýšit použitelnost této aplikace a zpříjemnit její užívání. Algoritmus pro vygenerování kompozice vestavného systému, který je popsán v Kapitole 6.4.1, generuje zařízení do n-úhelníku dle zadání v Kapitole 6.1.1. Pro přehlednější a konformnější ovládání je žádoucí, aby si uživatel sám rozvrhnul rozmístění prvků vestavného systému. Při přetahování se canvas podbarví zelenou barvou. Toto podbarvení značí, že přetahování je aktivované a po dokončení bude mít prvek novou pozici. Aktivování přetahování znamená, že byly překročené minimální hranice pro vertikální a horizontální vzdálenost, které jsou definovány OS Windows.

Uživatel má na výběr ze dvou akcí:

1. **Přetahování zařízení** — tato akce slouží pro rozložení zařízení po canvasu. Po dokončení přetahování bude mít zařízení novou pozici a automaticky se přepočítají pozice všech komunikačních kanálů a popisů, které se vztahují k tomuto zařízení.

2. **Přetahování komunikačních kanálů** — tato akce slouží pro rozložení komunikačních kanálů po canvasu. Standartní generování komunikačních kanálů vychází z toho, že komunikační kanál začíná a končí ve středu zařízení. Při více stejných komunikačních spojení nebo opačných spojení se komunikační kanály překrývají a je nutné tuto situaci zohlednit. Přetahování komunikačních kanálů se provádí tažením místa pro token (střed komunikačního kanálu kruhovitého tvaru s bílou výplní). Po přetažení bude mít kanál novou pozici a všechna zařízení zůstávají na stejné pozici.

6.9.1 Uložení rozložení kompozice vestavného systému

Rozložení kompozice vestavného systému se dá uložit do XML souboru, který obsahuje X a Y souřadnice všech vestavných zařízení a komunikačních kanálů. Uložení se provádí pomocí klávesové zkratky `CTRL+S` nebo přes `RibbonMenu -> Save`. Rozložení se uloží do souboru s implicitním názvem layout.xml. Nebo se kompozice dá také `Uložit jako` pomocí klávesové zkratky `CTRL+SHIFT+S` nebo přes `RibbonMenu -> Save As`. Zobrazí se dialog pro uložení XML souboru a uživatel si může soubor pojmenovat a uložit dle libosti.

6.9.2 Načtení rozložení kompozice vestavného systému

Načtení XML souboru, který popisuje rozložení kompozice vestavného systému se provádí pomocí klávesové zkratky `CTRL+O` nebo přes `RibbonMenu -> Open`. Po vybrání existujícího XML souboru se zavolá privátní metoda `LoadXMLLayout(string fileName)`, která načte vybraný soubor a rozmístí zařízení a všechny prvky s nimi spjaté do souřadnic, které jsou definované v XML souboru. Metoda má návratový typ `void`.

6.9.3 Struktura XML souboru

XML soubor obsahuje kořenový element `EmbeddedSystem`, který obsahuje elementy `Devices` a `ComunicationChannels`. Souřadnice X, Y jsou datového typu `Double` a mají referenční počátek v levém horním rohu aplikace.

```
<?xml version="1.0" encoding="utf−8"?>
<EmbeddedSystem>
  <Devices>
    <Device name="Dev0">
      <x>439</x>
      <y>236.04</y>
    </Device>
   .
   .
   <Device name="InCOIN">
      <x>649</x>
      <y>288.04</y>
    </Device>
  </Devices>
  <ComunicationChannels>
    <ComunicationChannel name="active">
```

```
        <x>261</x>
        <y>12.04</y>
      </ComunicationChannel>
      .
      .
      <ComunicationChannel name="soldItem">
        <x>365</x>
        <y>55.04</y>
      </ComunicationChannel>
    </ComunicationChannels>
</EmbeddedSystem>
```

Výpis 7: Struktura XML souboru rozložení kompozice vestavného systému

7 Ukázka vygenerovaného příkladu

Tato kapitola je věnována příkladu pro řízení vestavného systému z kapitoly 4. Po spuštění překladače s breakpointy se vygeneruje a spustí simulátor. Kompozice vestavného systému je rozmístěna do šestiúhelníku viz. obrázek 22. Pro lepší přehlednost je nutné rozložit zařízení technikou drag-and-drop nebo načíst rozložení z XML souboru viz. obrázek 23. Struktura rozložení je v příloze A jako výpis 10.

A nyní se může rozběhnout simulace vestavného systému klávesovou zkratkou `F5` nebo přes menu ikonou `Debug` v záložce `Debug`. Simulace zatím pořád stojí, protože čeká na akci uživatele (musí se vhodit dostatečná částka a zvolit předmět k prodeji). Uživatel si zvolí předmět číslo 2 (modelový příklad obsahuje pouze předměty s číslem 0 - 2). Volba předmětu se provádí dvojklikem na vstupní zařízení s názvem `InINPUT`. Tato volba představuje panel automatu pro výběr předmětu.

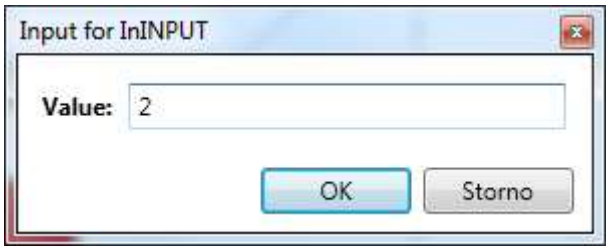

Figure 20: Dialog pro výběr předmětu

Po zvolení předmětu se se hodnota zapíše do komunikačního kanálu s názvem `input` a simulace se automaticky spustí. Tuto hodnotu přijímá zařízení s názvem `Dev1`, které vybírá proces s názvem `selection`. Uživatel je informován, že si vybral předmět s číslem 2. Informování uživatele se provádí pomocí zařízení s názvem `Dev0`, které představuje display automatu. Zařízení s názvem `Dev1` předá komunikačním kanálem `load` zprávu o připravení zvoleného předmětu uživatelem, zjistí se dostupnost předmětu. V případě, že je předmět dostupný vrací se zpráva o částce předmětu po komunikačním kanálu `selectedItem` řízení zařízení s názvem `Dev1`, které vybírá proces s názvem `controllerProcess`. Řízení je předáno zařízení s názvem `Dev2` po komunikačním kanálu `choice`. Zařízení vybírá proces s názvem `serve` a čeká na zaplacení celé částky předmětu (v tomto případě 15 Kč). Simulace se opět zastaví a čeká se na vhození mincí. Uživatel postupně vhazí mince s monimální hodnotou 10, 2, 2 a 1 Kč. Vhození mince se opět provádí dvojklikem na vstupní zařízení s názvem `InCOIN`, které představuje otvor pro vhození mince do automatu.

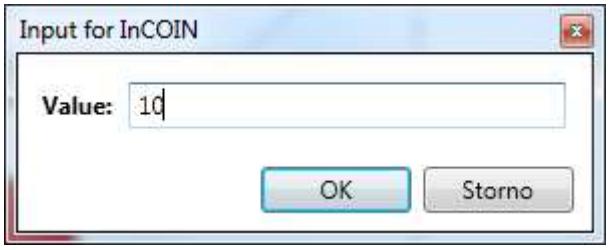

Figure 21: Dialog pro vházení mincí

Po každém vhození se hodnota zapíše do komunikačního kanálu s názvem `coin` a simulace pokračuje. Zařízení s názvem `Dev3` přijímá tuto hodnotu a vybírá proces s názvem `count`. Tento proces počítá vhozenou částku. V případě, že se vhozená částka rovná požadované částce předmětu předá se řízení zpátky zařízení s názvem `Dev2` po komunikačním kanálu `paid`. Tato akce se dá považovat za vydání požadovaného předmětu uživateli. Pro odečtení předmětu z databáze automatu je nutné ještě celý proces korektně ukončit. Proto zařízení s názvem `Dev2` nyní vybírá proces s názvem `paid` a posílá informaci o zaplací dále po komunikačním kanálu s názvem `soldItem`. Zařízení s názvem `Dev2` vybírá proces s názvem `served` a posílá zprávu dále po komunikačním kanálu názvem `remove` zařízení s názvem `Dev4`, ať tento předmět odstraní ze své databáze. Zažízení s názvem `Dev4` vybírá proces `removeItem` a příslušný předmět odečítá ze své databáze. Nakonec je uživatel informován, že předmět je prodaný. Automat je nyní připraven obsloužit dalšího zákazníka.

V případě, že je požadovaný předmět prodaný a zákazník si zvolí tento předmět je proces zkrácený. Po zvolení předmětu se proces vykováná stejně až do kontroly dostupnosti v databázi zařízením `Dev4`. Po komunikačním kanálu `selectedItem` vrací, že předmět je vyprodaný a zařízení `Dev1` posílá po komunikačním kanálu `diplayInput1` zprávu pro informování uživatele, že si vybral vyprodaný předmět. Po zobrazení této zprávy na displayi se proces ukončí.

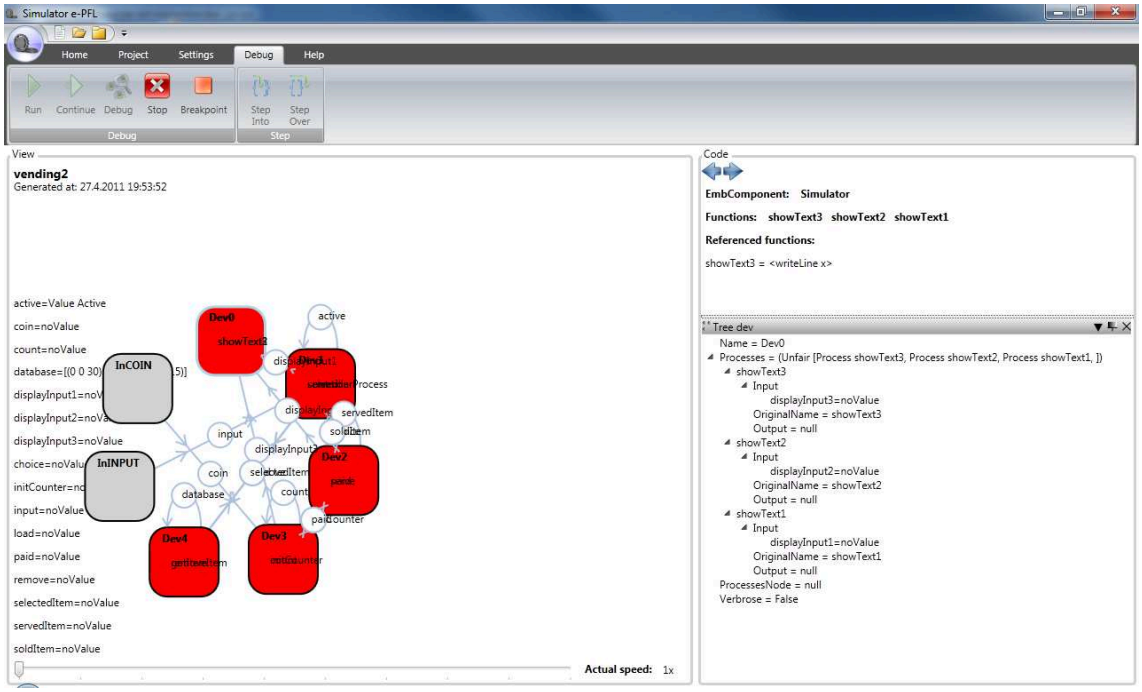

Figure 22: Dialog pro nastavení breakpointů

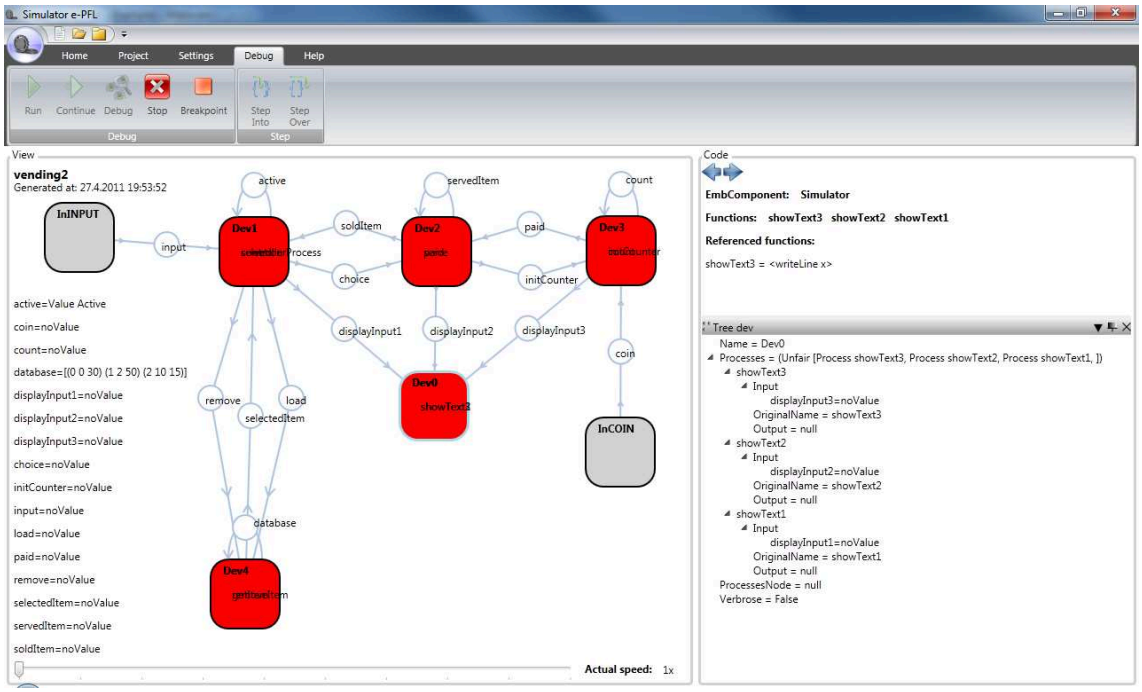

Figure 23: Dialog pro nastavení breakpointů

8 Závěr

Cílem této diplomové práce byla implementace simulátoru, dále rozšíření a úprava překladače pro generování konfigurace *PFL* do WPF aplikace s objektovým jazykem C#. Simulátor e-*PFL* je určitě přínosem pro testování vestavných systémů na úrovni koordinační vrstvy. Do této doby byla k dispozici pouze konzolová aplikace, která jen vypisovala informace na obrazovku. V simulátoru se ověřuje korektní sestavení a funkce vestavných systémů, v případě nepředvídaného chování se dá chod vestavných systémů ladit přímo v simulátoru a po opravě se spustí simulace znovu. Odpadá zde nutnost z návrhu vestavného systému realizovat tento systém a po testování ve zkušebním provozu se zjistí, že se stala chyba v návrhu a celý cyklus je nutno opakovat. Zejména se zde šetří čas a náklady potřebné pro realizaci vestavného systému.

Mým osobním přínosem je, že jsem se naučil technologii WPF, kterou budu nadále používat místo dosavadní technologie WinForms. WPF je docela podobná platformě Silverlight, která se používá pro vývoj RIA.

Simulátor se dá dále rozšiřovat o diagnostické funkce pro vestavný systém. Funkce by mohly sledovat zařízení a při neočekávaném průběhu či poruše systému by se uživatel dozvěděl o vzniklém problému. Nebo by se mohl každý stav systému ukládat do historie a kdykoliv by se mohl aktuální stav systému vrátit do uloženého stavu z historie a pokračovat dále ve vykonávání.

9 Reference

[1] Běhálek, Marek. *Vestavný procesně funkcionální jazyk, Autoreferát disertační práce*. Ostrava, 2010.

[2] Thompson, D.: *Embedded Programming with the Microsoft .NET Micro Framework (Pro - Developer)*. Redmond,WA, USA: Microsoft Press, 2007, ISBN 0735623651.

[3] Vahid, F.; Givargis, T.: *Embedded System Design: A Unified Hardware/Software Introduction*. New York, NY, USA: JohnWiley & Sons, Inc., 2001, ISBN 0471386782.

[4] Wallace, M.; Runciman, C.: Extending a functional programming system for embedded applications. *Softw. Pract. Exper.*, ročník 25, č. 1, 1995: s. 73–96, ISSN 0038-0644.

[5] Huch, F.: Learning programming with erlang. In *ERLANG '07: Proceedings of the 2007 SIGPLAN workshop on ERLANG Workshop*, New York, NY, USA: ACM, 2007, ISBN 978-1-59593-675-2, s. 93–99.

[6] Loidl, H.-W.; Rubio, F.; Scaife, N.; aj.: Comparing Parallel Functional Languages: Programming and Performance. *Higher Order Symbol. Comput.*, ročník 16, č. 3, 2003: s. 203–251, ISSN 1388-3690.

[7] Peyton Jones, S.; Gordon, A.; Finne, S.: Concurrent Haskell. In *POPL '96: Proceedings of the 23rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, New York, NY, USA: ACM, 1996, ISBN 0-89791-769-3, s. 295–308.

[8] Halbwachs, N.; Caspi, P.; Raymond, P.; aj.: The synchronous dataflow programming language LUSTRE. In *Proceedings of the IEEE*, 1991, s. 1305–1320.

[9] Hammond, K.; Michaelson, G.: Hume: A Domain-Specific Language for Real-Time Embedded Systems. In *Generative Programming and Component Engineering, Second International Conference*, GPCE 2003, Erfurt, Germany, September 22-25, 2003, Proceedings, Lecture Notes in Computer Science, ročník 2830, editace F. Pfenning; Y. Smaragdakis, Springer, 2003, s. 37–56.

[10] Specht, E.; Redin, R. M.; Carro, L.; aj.: Analysis of the use of declarative languages for enhanced embedded system software development. In *SBCCI '07: Proceedings of the 20th annual conference on Integrated circuits and systems design*, New York, NY, USA: ACM, 2007, ISBN 978-1-59593-816-9, s. 324–329.

[11] Nyström, J.; Trinder, P.; King, D.: Evaluating high-level distributed language constructs. In *ICFP '07: Proceedings of the 12th ACM SIGPLAN international conference on Functional programming*, New York, NY, USA: ACM, 2007, ISBN 978-1-59593-815-2, s. 203–212.

[12] Martin, G.: UML for Embedded Systems Specification and Design: Motivation and Overview. *Design, Automation and Test in Europe Conference and Exhibition*, ročník 0, 2002: str. 0773.

[13] Patai, G.; Hanák, P.: Embedded Functional Programming in Hume. In *IASTED on Software Engineering*, Innsbruck, Austria: ACTA Press, 2007, s. 328–333.

[14] Jones, N. D.; Gomard, C. K.; Sestoft, P.: *Partial evaluation and automatic program generation*. Upper Saddle River, NJ, USA: Prentice-Hall, Inc., 1993, ISBN 0-13-020249-5.

[15] Kollár, J.; Porubän, J.; Václavík, P.: From Eager PFL to Lazy Haskell. *Computers and Artificial Intelligence*, ročník 25, č. 1, 2006

[18] Woodcock, J.: *Slovník výpočetní techniky*. Praha, Microsoft Press 1993. ISBN 80-85297-48-5.

A Zdrojové kódy

A.1 e-*PFL*

Ukázkový příklad pro vending machine.

```
import "Prelude.pfl"
data State = Active
———————— JUST FOR TESTING
testInputData = [0,2,2,2,1,1,1,1]
testInput :: a Integer -> (a Integer, input Integer) testInput x = (x+1, (x % (length testInputData)) !! testInputData)
testInputDevice :: Device testInputDevice = Process testInput
testInputData2 = [5,5,10,10,20,2,2,5]
testInput2 :: b Integer -> (b Integer, coin Integer) testInput2 x = (x+1, (x % (length testInputData2)) !! testInputData2)
testInputDevice2 :: Device testInputDevice2 = Process testInput2
test1 :: coin Integer->Bool test1 coin = if coin == 50 then False else True
———————— MEMORY
memoryData = [(0,0,30),(1,2,50),(2,10,15)]
getItem :: load Integer -> database [(Integer, Integer, Integer)] -> (database [(Integer, Integer, Integer)], selectedItem (Integer,Integer, Integer))
getItem input database = let choice = input !! database in (database, choice)
removeItem :: remove Integer -> database [(Integer, Integer, Integer)] -> (database [(Integer, Integer, Integer)])
removeItem x database = let choice = x !! database ls = take x database rs = drop (x+1) database in case choice of {(a,b,c) -> (ls ++ [(a,b-1,c)] ++ rs)}
memory :: Device memory = Unfair [Process removeItem, Process getItem]
———————— CONTROLER
selection :: input Integer -> active EmbValue State -> (load Integer, displayInput1 [Character]) selection input active = (input, "Selected: "++show input)
controllerProcess :: selectedItem (Integer,Integer, Integer) -> (displayInput1 [Character], active EmbValue State, choice EmbValue (Integer, Integer)) controllerProcess choice = case choice of {(a,0,c) -> ("No more items: "++show a, Value Active, NoValue);
(a,b,c) -> ("Chosen: "++show a, NoValue, Value (a,c))}
served :: soldItem Integer -> (remove Integer, displayInput1 [Character], active EmbValue State) served x = (x, "Served: "++show x, Value Active)
```

```
controller :: Device controller = Fair [Process selection, Process served, Process controllerProcess]
———————— SERVING
serving :: Device serving = Fair[Process serve, Process paid]
serve :: choice EmbValue (Integer, Integer) -> (servedItem Integer, initCounter EmbValue Integer, displayInput2 [Character]) serve x = case x of {Value (a,b) -> (a, Value b, "Serving item: "++show a++", price: "++show b)}
paid :: servedItem Integer -> paid EmbValue State -> (soldItem Integer, displayInput2 [Character]) paid x y = (x, "Served: "++show x)
———————— MONEY
counter :: Device counter = Unfair [Process initCounter,Process count]
initCounter :: initCounter EmbValue Integer -> (count EmbValue Integer, displayInput3 [Character]) initCounter x = (x, "Conter initiated: "++show x)
count :: count EmbValue Integer -> coin Integer -> (paid EmbValue State, count EmbValue Integer, displayInput3 [Character]) count x y = case x of {(Value a) -> if (a - y) > 0 then (NoValue, Value (a-y), " Counter: "++show a++" Recieve coin: "++show y) else (Value Active, NoValue, " Counter: "++show a++" Recieve coin: "++show y++" Paid.")}
———————— PRINTER
showText1 ::displayInput1 [Character] -> () showText1 x = writeLine x
showText2 ::displayInput2 [Character] -> () showText2 x = writeLine x
showText3 ::displayInput3 [Character] -> () showText3 x = writeLine x
printer :: Device printer = Unfair [Process showText3, Process showText2,Process showText1]
————————— JUST INITIAL CONDITIONS
setA :: a Integer -> () setA x = ()
setB :: b Integer -> () setB x = ()
setActive :: active EmbValue State -> () setActive x = ()
setData :: database [(Integer, Integer, Integer)] -> () setData x = ()
main = (setData memoryData)`bl`(setActive (Value Active))`bl`(setA 0)`bl`(startDevice printer Simulator [])`bl`(startDevice controller Simulator [])`bl`(startDevice serving Simulator [])`bl`(startDevice counter Simulator [])`bl`(startDevice memory Simulator [])
```

Výpis 8: Definice vestavného systému vending machine

A.2 C#

Metoda `Run` pro vykonávání běhu zařízení.

```
protected void Run()
{
    if (Verbrose) Console.WriteLine("Dev " + Name + ": started - " + processes);
    while (true)
    {
        if (StepOverDebug)
        {
            IsWorking = false;
            Thread.CurrentThread.Suspend();
        }
        if (StepIntoDebug)
        {
            IsWorking = false;
            Thread.CurrentThread.Suspend();
        }
        IsWorking = true;
        Thread.Sleep((int)(SpeedDevice / DeviceManager.Sleep) / DeviceManager.STATECOUNT);
        //najde proces co je na rade
        ProcessNode node = processes.FindProcessToExecute();
        ExecutedProcess = node;
        if (node != null)
        {
            if (StepIntoDebug)
            {
                IsWorking = false;
                Thread.CurrentThread.Suspend();
            }
            IsWorking = true;
            DeviceState = EDeviceState.Waiting;
            Thread.Sleep((int)(SpeedDevice / DeviceManager.Sleep) / DeviceManager.
                STATECOUNT);
            //vetev kdyz nasel
            if (Verbrose) Console.WriteLine("Dev " + Name + ": selected - " + node);
            //proces bude vykonavan, budou na nej aplikovany hodnoty, proto je klonovan
            object execution = node.Process.Clone();
            if (StepIntoDebug)
            {
                IsWorking = false;
                Thread.CurrentThread.Suspend();
            }
            IsWorking = true;
```

```
DeviceState = EDeviceState.Processing;
Thread.Sleep((int)(SpeedDevice / DeviceManager.Sleep) / DeviceManager.
    STATECOUNT);
if (Verbrose) node.WriteStartMessage();
foreach (EmbeddedVariable input in node.Input)
{
    //konzumuje promenne
    execution = ((Function) execution).Apply(input.Consume(Verbrose));
}
if (execution is Constructor && ((Constructor) execution).Name == Constants.
    UNIT_CONS_NAME)
{
    if (Verbrose) node.WriteEndMessage();
    if (StepIntoDebug)
    {
        IsWorking = false;
        Thread.CurrentThread.Suspend();
    }
    DeviceState = EDeviceState.Idle;
    continue;
}
if (node.Output == null) throw new RuntimeException("Output␣of␣the␣runable␣
    process␣undefined.");

if (node.Output.Length == 1)
{
    //ceka na uvolneni
    node.Output[0].CanUpdate(Verbrose);
    // volny kanal naplni
    node.Output[0].ForceUpdate(execution, Verbrose);
    if (Verbrose) node.WriteEndMessage();
    if (StepIntoDebug)
    {
        IsWorking = false;
        Thread.CurrentThread.Suspend();
    }
    DeviceState = EDeviceState.Idle;
    continue;
}
if (execution is Constructor && ((Constructor) execution).Name.StartsWith(Constants.
    TUPLE_CONS_NAME))
{
    //pokud je jich vice, ntice, otestuje nejpre vsechny
    for (int i = 0; i < node.Output.Length; i++)
```

```
                {
                    if (((Constructor)execution).Args(i) is Constructor && ((Constructor)((
                        Constructor)execution).Args(i)).Name == Constants.NOVALUE) continue
                        ;
                    node.Output[i].CanUpdate(Verbrose);
                }
                // pak vsechny zapise
                for (int i = 0; i < node.Output.Length; i++)
                {
                    if (((Constructor)execution).Args(i) is Constructor && ((Constructor)((
                        Constructor)execution).Args(i)).Name == Constants.NOVALUE) continue
                        ;
                    node.Output[i].ForceUpdate(((Constructor)execution).Args(i), Verbrose);
                }
                if (Verbrose) node.WriteEndMessage();
                if (StepIntoDebug)
                {
                    IsWorking = false;
                    Thread.CurrentThread.Suspend();
                }
                DeviceState = EDeviceState.Idle;
                continue;
            }
            throw new RuntimeException("Internal␣error:␣Unexpected␣output.");
        }
    }
}
```

Výpis 9: Změna vnitřního stavu zařízení

A.3 XML

Ukázkový příklad XML souboru pro načtení rozvržení kompozice vestavného systému vending machine.

```
<?xml version="1.0" encoding="utf-8"?>
<EmbeddedSystem>
  <Devices>
    <Device name="Dev0">
      <x>439</x>
      <y>236.04</y>
    </Device>
    <Device name="Dev1">
      <x>236</x>
```

```
      <y>62.04</y>
    </Device>
    <Device name="Dev2">
      <x>442</x>
      <y>62.04</y>
    </Device>
    <Device name="Dev3">
      <x>650</x>
      <y>61.04</y>
    </Device>
    <Device name="Dev4">
      <x>226</x>
      <y>448.04</y>
    </Device>
    <Device name="InINPUT">
      <x>39</x>
      <y>46</y>
    </Device>
    <Device name="InCOIN">
      <x>649</x>
      <y>288.04</y>
    </Device>
  </Devices>
  <ComunicationChannels>
    <ComunicationChannel name="active">
      <x>261</x>
      <y>12.04</y>
    </ComunicationChannel>
    <ComunicationChannel name="coin">
      <x>674.5</x>
      <y>199.54</y>
    </ComunicationChannel>
    <ComunicationChannel name="count">
      <x>675</x>
      <y>11.04</y>
    </ComunicationChannel>
    <ComunicationChannel name="database">
      <x>251</x>
      <y>398.04</y>
    </ComunicationChannel>
    <ComunicationChannel name="displayInput1">
      <x>362.5</x>
      <y>174.04</y>
    </ComunicationChannel>
```

```
<ComunicationChannel name="displayInput2">
  <x>465.5</x>
  <y>174.04</y>
</ComunicationChannel>
<ComunicationChannel name="displayInput3">
  <x>569.5</x>
  <y>173.54</y>
</ComunicationChannel>
<ComunicationChannel name="choice">
  <x>363</x>
  <y>115.04</y>
</ComunicationChannel>
<ComunicationChannel name="initCounter">
  <x>573</x>
  <y>116.04</y>
</ComunicationChannel>
<ComunicationChannel name="input">
  <x>162.5</x>
  <y>79.02</y>
</ComunicationChannel>
<ComunicationChannel name="load">
  <x>303</x>
  <y>252.04</y>
</ComunicationChannel>
<ComunicationChannel name="paid">
  <x>572</x>
  <y>56.04</y>
</ComunicationChannel>
<ComunicationChannel name="remove">
  <x>212</x>
  <y>252.04</y>
</ComunicationChannel>
<ComunicationChannel name="selectedItem">
  <x>257</x>
  <y>272.04</y>
</ComunicationChannel>
<ComunicationChannel name="servedItem">
  <x>467</x>
  <y>12.04</y>
</ComunicationChannel>
<ComunicationChannel name="soldItem">
  <x>365</x>
  <y>55.04</y>
</ComunicationChannel>
```

```
    </ComunicationChannels>
</EmbeddedSystem>
```

Výpis 10: XML s rozvržením kompozice vestavného systému